U.S.N.A. — Trident Scholar project report; no. 290 (2002)

SUPPORTING SECURE, AD HOC JOINS FOR TACTICAL NETWORKS

by

Midshipman Joshua B. Datko, Class of 2002
United States Naval Academy
Annapolis, Maryland

Certification of Advisers Approval

Assistant Professor Margaret M. McMahon
Computer Science Department

Associate Professor Donald M. Needham
Computer Science Department

Acceptance for the Trident Scholar Committee

Professor Joyce E. Shade
Deputy Director of Research & Scholarship

USNA-1531-2

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATE COVERED
	7 May 2002	

4. TITLE AND SUBTITLE
Supporting secure, ad hoc joins for tactical networks

5. FUNDING NUMBERS

6. AUTHOR(S)
Datko, Joshua B.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)		8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER
US Naval Academy Annapolis, MD 21402		Trident Scholar project report no. 290 (2002)

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
This document has been approved for public release; its distribution is UNLIMITED.	

13. ABSTRACT:

Modern warfare tactics demand timely, high quality intelligence information. Strike aircraft are in special need of accurate, real-time targeting information due to their proximity to hostile tar-gets. The Defense Advanced Research Project Agency's Tactical Targeting Network Technology (TTNT) initiative focuses on responding to this need by improving distributed command and control operations through a low-latency, high bandwidth, and dynamically reconfigurable network infrastructure. In this research, we develop an algorithm that supports the entry of a TTNT participant into a pre-existing, ad hoc, and wireless net-centric environment. Analysis of the shortcomings of similar current technologies, specifically Jini networking technology and Bluetooth, established a need for a security-focused approach to ad hoc networking. Likewise, popular secure Public Key Infrastructure (PKI) implementations have also proven insufficient due to their reliance on non-mobile systems. The algorithm presented in this project applies a novel key management procedure to provide information assurance in the TTNT realm. The implementation of the key management scheme included the creation of a simulation to test different network joining scenarios. This simulation provided both a successful implementation of the secure joining algorithm, as well as the means to collect empirical runtime measurements. Incorporation of a trust management scheme is also discussed. Our approach addresses the complex scenarios in which a previously authenticated network node could verify a joining user's credibility. This research provides a necessary first step in the development of ad hoc networks suitable for employment in network centric warfare operations. We demonstrate the capability for wireless nodes to rapidly and securely join existing TTNT networks. Additionally, this research provides a key management approach that contributes to the design of secure, ad hoc networks.

14. SUBJECT TERMS	15. NUMBER OF PAGES
tactical targeting network technology, ad hoc networking, net-centric environment	87
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Abstract

Modern warfare tactics demand timely, high quality intelligence information. Strike aircraft are in special need of accurate, real-time targeting information due to their proximity to hostile targets. The Defense Advanced Research Project Agency's Tactical Targeting Network Technology (TTNT) initiative focuses on responding to this need by improving distributed command and control operations through a low-latency, high bandwidth, and dynamically reconfigurable network infrastructure. In this research, we develop an algorithm that supports the entry of a TTNT participant into a pre-existing, *ad hoc*, and wireless net-centric environment.

Analysis of the shortcomings of similar current technologies, specifically Jini networking technology and Bluetooth, established a need for a security-focused approach to *ad hoc* networking. Likewise, popular secure Public Key Infrastructure (PKI) implementations have also proven insufficient due to their reliance on non-mobile systems. The algorithm presented in this project applies a novel key management procedure to provide information assurance in the TTNT realm. The implementation of the key management scheme included the creation of a simulation to test different network joining scenarios.

This simulation provided both a successful implementation of the secure joining algorithm, as well as the means to collect empirical runtime measurements. Incorporation of a trust management scheme is also discussed. Our approach addresses the complex scenarios in which a previously authenticated network node could verify a joining user's credibility.

This research provides a necessary first step in the development of ad hoc networks suitable for employment in network centric warfare operations. We demonstrate the capability for wireless nodes to rapidly and securely join existing TTNT networks. Additionally, this research provides a key management approach that contributes to the design of secure, *ad hoc* networks.

Acknowledgements

To those that have supported me in this effort (academically, emotionally, physically, etc. . .):

- To my advisors, Assistant Professor Margaret McMahon and Associate Professor Donald Needham, thank you for the challenge, your patience, and for taking the "dead" out of "deadline."

- To the Computer Science Department at the United States Naval Academy, *every* faculty/staff member has provided encouragement or support in some way. I hope I experience another group of faculty as devoted to teaching and the pursuit of academia.

- To Monty Python, Douglas Adams, Neal Stephenson, and Bruce Schneier, thank you for providing great works that led me down the road of humor, humor/science, science/cryptography, and cryptography respectfully.

- To Farrah, thank for the constant well of love, encouragement, and support. Thank you also for being the co-expert in this subject.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The emerging Tactical Targeting Network Technology (TTNT) [25] is a realization of the Network-Centric Warfare (NCW) concept that seeks to provide the backbone for disseminating tactical data within a networked battle environment [1]. The NCW concept defines a networking system composed of a series of grids that, when combined, allow a heightened degree of battlespace awareness. A NCW system attains this increased awareness by distributing the command and control entities across the network. The focus of TTNT is to achieve the NCW warfare mission through the establishment of a distributed, dynamically reconfigurable network. Such a network is expected to aid every warfighting entity, especially tactical aircraft, due to the high-speed nature of tactical aviation environments. With TTNT, aircraft promulgate targeting information through *ad hoc* connections to improve reaction time to pop-up targets.

The TTNT network needs to be distributed, net-centric, and secure. Security in such a networking environment is complex due to the rapid mobility, inherent uncertainty, and the power constraints of the networking entities. A reliance on trusted third parties, as most authentication schemes currently use, does not incorporate well into the TTNT environment. The standard version of the Kerberos authentication protocol [24] relies on a trusted third party for every authentication. Such an approach in the TTNT domain would inhibit the desired flexibility and distributed nature due to the constant references to this static third party.

Current security practices that include frequent trusted third party referral and the ability to perform high computation public-key actions do not function properly in the *ad hoc*, wireless domain. Most mobile entities are power-constrained, and consequently have relatively low computational power available. Also, unlike conventionally wired networks, physical security of the transport media is not as easily controlled within the wireless network domain. Since wireless transmission occurs through the open air, wireless networks receive greater exposure to eavesdropping, denial of service attacks, and other malignant network behavior [18]. Establishing secure communications in a dynamic environment is further challenged when the communication entities do not hold any *a priori* information about each other.

1.1 Motivation

As of this writing, the United States has incorporated network centric technology in recent operations in Afghanistan. Currently, a successful NCW implementation example involves the RQ01 Predator Unmanned Aerial Vehicle (UAV). A Predator is equipped with the capability to send live video feed to an AC-130H Spectre Gunship. This pairing was effectively used in a recent attack near the Zawar Kili cave complex in Eastern Afghanistan and demonstrates the utility of net-centric technology [17]. The Predator was able to obtain real-time video surveillance to assess the target, enabling the AC-130 to engage the target upon arrival. Previous AC-130 tactics required that the aircraft first over fly the target to obtain the necessary targeting data. This not only alerts the target to the eminent attack, but also extends the exposure of the aircraft to counterattack. Although the AC-130 now possesses a Predator link, other aircraft have yet to receive this modification, resulting in ground operators still talking the pilots onto their targets [17].

Although UAVs provide real-time intelligence, the heart of network centric operations lies in giving targeting information directly to the pilot, soldier, or sailor. However, other platforms may benefit from direct tactical sensory data, such as a targeting solution from an attack aircraft's radar. If numerous platforms have the potential to share targeting information, a network technology is needed to provide, manage, and maintain a high level of

connectivity. Specific network connections, such as those in the AC-130/Predator combination, increase the battlespace awareness only for the specialized platform. What is needed is a network standard like TTNT to share data among all platforms in the battlespace.

1.2 Organization of Report

The remainder of this report is organized as follows: Section 2 discusses the framework technologies and concepts that are applied in the research. In Section 3 we present the joining algorithm and the assumptions concerning the initial conditions. Section 4 describes the prototype simulations and proof of concept. Section 5 discusses the actual implementation and final design considerations. Section 6 describes the experimental procedure and the application to TTNT scenarios. In Section 7 we draw our analysis from the experiments, and in Section 8 we present our conclusions.

In Appendix A we include a discussion of our progress throughout the course of the project compared to our proposed timeline. Appendices B and C are the Java source files `AuthenticationClient.java` and `AuthenticationService.java` respectfully, that outline our implementation discussed in Section 5. Lastly, Appendix D contains instructions for installation and use of the simulation software.

Chapter 2

Background

In this section we present the fundamental principles and technologies underlying our work. Section 2.1 describes the original motivation for Tactical Targeting Network Technology (TTNT) and defines the scope of this project. In Section 2.2, we discuss key Network Centric Warfare concepts and apply them to TTNT. In Section 2.3 we provide a summary of the underlying public key cryptography and digital certificate technology of an authentication algorithm. Section 2.4 contains an analysis of mobile, *ad hoc* networking security and the inherent security difficulties that typically arise when designing and using such a network. Finally, in Section 2.5, we present two current *ad hoc* networking technologies and evaluate both technologies from the perspective of meeting the needs of the TTNT specifications.

2.1 TTNT Design Goals

One of the main motivating factors for DARPA's TTNT initiative is the need to respond to targets of opportunity with greater speed [25]. This technology is meant to allow tactical aircraft to immediately share fire-control and battle damage assessment data of a pop-up target, such as an unanticipated surface-to-air missile site. Designers are currently working to produce a complete product, to include the construction of TTNT hardware. The TTNT program is concerned with frequency hopping schemes, error correcting protocols, and supporting coexistence with LINK-16 systems. Eventually, TTNT will also incorporate the flow

of sensitive, tactical fire-control information.

The algorithm developed as a result of this Trident research focuses on a limited subset of the TTNT problem space. Specifically, the algorithm presents a solution to the problem of joining a network in an *ad hoc*, network-centric manner. The actual TTNT will have to address issues involving situations in the absence of a pre-existing network, or where many nodes attempt to join at once. The fundamental assumption of our algorithm is that an entity desires to join an established *ad hoc* network of more than one node. This assumption reduces the complexity of the problem to research, develop, and test the algorithm within the Trident time-line.

2.2 TTNT as a Network-Centric Infrastructure

Two key components of TTNT are computational grids and net-centricity [15]. By way of comparison, consider an electric power infrastructure that provides clients with instantaneous and convenient access, enabling end-users to power a myriad of devices. Likewise, a computational grid attempts to provide computing resource access to its clients. A computational grid is a computing infrastructure that provides pervasive and inexpensive access to networked entities [15]. The grid forces operations away from the individual platform and toward the networked entities. Critical aspects of net-centricity include the ability of a networking infrastructure to provide services on-demand, and the ability of the network to be self-administering. Note that a large collaboration of platforms, like the Internet, is not inherently net-centric. Such a network merely allows computational entities to physically connect to each other. In a net-centric system, the network itself is the computational entity. A net-centric system can be expected to serve the user more efficiently through its ability to manage its own resources [10].

To establish net-centricity, each NCW grids decomposes into a set of three specialized grids: the information grid, the sensor grid, and the engagement grid [15]. The information grid is a networking hardware construct that provides the mechanism for data communication. It also facilitates an interface for its users to access the information they need. The

information grid is dynamic, and provides capabilities for properly adapting its resource allocation as users and servers join and leave the network. The information grid provides an infrastructure that can administer its own resources, a concept known as *self-synchronization* [1].

As depicted in Fig. 2.1, there are three entities that compose the net-centric grid model: sensors, shooters, and decision makers. The first, and most prominent, are sensor entities. Sensor entities collect data, and may aid in the distribution of other sensory data. In the NCW model, a federation of sensors installed in a computation grid is called the sensor grid. The second type of NCW entity is a shooter, which are tactical platforms with fire-control capabilities, such as a fighter aircraft. Shooters reside mainly on the engagement grid, however a tactical platform may also have sensory devices. Thus, a platform may be a sensor and a shooter, participating in both grids. This is represented in Fig. 2.1 by the diamond figure, with the circle inside to depict a shooter with a TTNT sensor installed.

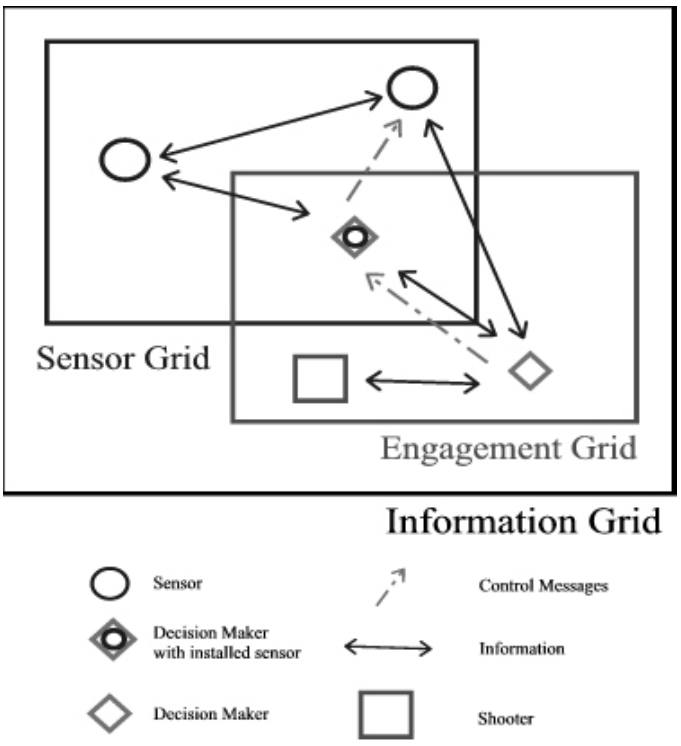

Figure 2.1: TTNT Logical Design

Shooter entities can distribute fire control information, process sensory information, and engage designated targets. The third entity type in the net-centric grid represent decision makers. Decision maker entities can process both sensory and fire control information, but they are the only entities able to interpret the data and choose actions. In a distributed manner, their decisions disseminate through the net-centric grid to control the sensors and shooters. Furthermore, due to the adaptive nature of NCW, platforms may have changing roles acting as a combination of entities to respond to dynamic situations.

2.3 Public Key Cryptography and Digital Certificates

To provide secure message traffic to other entities on the grid, the communication protocol must employ some form of cryptography. Most modern cryptosystems either use symmetric or public key algorithms [21]; both are based on a key, or keys, that will handle encryption and decryption. Both public key and symmetric cryptosystems can provide properties of authentication, integrity, and non-repudiation in messages. Authentication ensures that the sender of the message has proven his identity to the receiver, while integrity states the message being transferred has not been altered enroute. Additionally, cryptography can provide non-repudiation, which prevents the sender from denying authorship of a sent message [21].

In symmetric cryptographic algorithms, both the sender and the receiver have identical keys. Since this key is used for both encryption and decryption, the entire security of the cryptosystem depends on the secrecy of the key [21]. Although symmetric algorithms can provide adequate security, distributing and managing keys is cumbersome. Since both parties require the same key to communicate, and the key must remain secret, the parties must be cautious when agreeing on their secret key. The safest method is an actual face-to-face meeting, or a transportation medium involving the highest levels of physical security. Key exchange is an inherent problem for symmetric key cryptography, because both parties must pre-arrange a means for distribution. The need for *a priori* knowledge in symmetric cryptosytems makes the ability to securely communicate with unknown individuals difficult.

Public key cryptography solves the failings of symmetric cryptography by using a private and public key pair. With this asymmetric cryptography approach, the private key cannot be calculated, within reasonable computational limits, if only the public key is known. The private key is kept secret and is used for decrypting a message, while the public key can be published or stored on a server and is needed for encryption. The Diffie-Hellman algorithm makes it possible to allow secure key distribution over an insecure channel [16]. Using the Diffie-Hellman key exchange algorithm, two individuals can communicate securely, without ever meeting or even knowing each other. Through application of the algorithm, two parties can calculate the same key, independent of each other, which allows the enabling of a link for secure traffic. Public key cryptography is not desirable for actual communication, but only for establishing communication by distributing symmetric keys [21]. Public key methods are used primarily for key distribution because they are more complex and require greater computational ability. Symmetric algorithms are significantly faster and more efficient for communication, therefore a public key algorithm will usually transmit a random key, called a session key, which will be used to communicate via symmetric encryption [21].

The ability to prove an identity is a critical component in Public Key Infrastructure (PKI), and digital signature protocols help to provide this authentication. In a manner similar to handwritten signatures, digital signatures provide a proof of authenticity to the receiver of a message. Based on public key cryptography, digital signatures rely on a public-private key pair. Further, digital signatures heavily employ the use of one-way hash functions. The one-way hash function, also known as a secure hash function, takes the message as input and computes a smaller, unique, fingerprint representation of the message. Any modulation of the message changes its hash value, and since the hash function is one-way, the message cannot be feasibly acquired from the hash [6].

After the sender hashes a message, it then will be signed using his private key. This signature is then sent attached to the message. Upon receipt, the receiver calculates the hash value of the message and, using the sender's public key, verifies the message. In order to achieve a successful verification, the hash value of the message on the receiving end must match the sender's original hash value. This confirms that the sender's private key

was actually used, and that the message remained unaltered during transfer [6]. Digital signatures not only provide authentication, but also provide another desirable aspect of public key infrastructure: non-repudiation [22]. Since a user's private key is unique, he is the only person that can sign his messages and thus cannot claim that he did not send the message.

The last main foundation of public key infrastructures that we address are digital certificates and the concept of trusted third parties. Digital certificates bind an identity to a public key pair to reduce the potential for public key impostors [5]. A digital certificate also contains a list of credentials about its owner, such as name, public key, and digital fingerprint. A trusted third party, or certification authority (CA), issues a certificate to a user, and then signs the certificate with the CA's private key. This eliminates some of the trust problems in the basic public key distribution methods. Since the real advantage of using a public key system is that two parties can establish secure communication over an insecure medium, the receiving party needs to have some assurance that the sender's public key does in fact belong to the sender. For example, if an individual receives a public key signed by a trusted CA and can verify the validity of the CA's signature on the key, then he can also have some confidence in the sender's public key.

Public key cryptography, digital signatures, one-way hash functions, and digital certificates, combined in a systematic manner, compose the foundations of a Public Key Infrastructure. A PKI manages CAs, but it also provides extra functionality through the ability to "issue, revoke, store, retrieve, and trust certificates" [12]. Since the PKI CA is the root authority for all issued certificates, its integrity and reliability is vital to all the certificates that it signs. The root CA is supported by the X.509 [12] certificate standard. An X.509 certificate is an identity certificate that contains information about the issuer that binds the certificate to a specific public key. The X.509 standard grants a PKI its hierarchal authority and provides it with a directory structure. Most X.509 implementations also prevent the user from owning multiple certificates. Therefore, each certificate holder is bound to one specific key that is provided by the CA.

2.4 Mobile *Ad Hoc* Network Security

Mobile, *ad hoc*, networks (MANETs) are inherently unstructured due to their dynamic topology [3]. Since entities continuously join and exit the network, the state of the network is in perpetual flux. This uncertainty, especially in a TTNT environment, dramatically increases the difficulty of securing the network. Security in such a networking environment is extremely complex due to the rapid mobility, inherent uncertainty, and power constraints of the *ad hoc* networking entities [18]. The security issues stem from the difficulty to establish and maintain a trust relationship in a MANET. *Ad hoc* connections need to occur quickly within the architecture; authentication is complex and resource intensive, especially considering MANETs' typical unstable topology. Various partial solutions exist in emerging MANETs, including the absence of security, simple shared-secret authentication, and full PKIs. Currently, standards such as the IEEE 802.11 standard for wireless Local Area Networks implement a symmetric key approach. This approach reduces the management complexity and encourages faster connection establishment. However, symmetric key solutions do not provide strong security and are easily susceptible to eavesdropping [20]. Conversely, PKI implementations in MANETs are cumbersome due to the inherent limitations of mobile computing. Public key algorithms are considerably more resource intensive than symmetric ones, which is an important consideration in light of the limited capabilities of MANET nodes. An obvious solution does not exist for supporting key management services like certification, revocation, and key issuance due to the numerous possibilities for insecurities. The limited capabilities of MANET nodes, the awkwardness of a PKI implementation, and the vulnerabilities of a symmetric key protocol result in highly suspect solutions to the issue of MANET security.

2.5 Bluetooth and Jini

Bluetooth [26] is an emerging wireless technology that applies an *ad hoc* connection scheme similar to that needed by TTNT. This technology allows many heterogeneous units to com-

municate in a localized *ad hoc* network. Bluetooth technology is ideal for home and small office networks that contain many computational devices, because it can offer wireless connectivity for each device. The proprietary Bluetooth algorithm for joining a network is based on a challenge-response authentication scheme [27]. Bluetooth devices use a symmetric key that is arranged *a priori*, with authentication being based on the challenger's ability to produce this key.

The underlying Bluetooth algorithms fail to support TTNT goals in a manner similar to the shortcomings of Sun Microsystems' Jini Technology [23]. Likewise, Jini does not include an authentication protocol, but instead relies on inherited security policies from Java. Currently, Sun Microsystems has incorporated enhanced authentication protocols to include Kerberos [24] network authentication support in Java 2 v1.4 [23]. Although Jini may be used with Java-based authentication, this implementation still lacks an adaptive approach to network joining management, as we further discuss in Section 5.2. Similarly, Bluetooth does not provide support for a grid-like infrastructure, and is not scalable to accommodate TTNT. Bluetooth's scalability is proximity based and is designed to only handle users in small groupings located within meters of each other.

Another shortcoming of Bluetooth is in its method of authentication. Its weakness lies in its assumption that the symmetric authentication key will remain secret among valid Bluetooth devices. For example, a Bluetooth device can masquerade as another device using that device's symmetric link key, thus destroying the trust relationship that is the basis for Bluetooth communication. While Bluetooth may be secure enough for small applications in the home, its ability to provide confidence on any large network with sensitive data is problematic [27]. Bluetooth's weak authentication scheme and lack of support for computational grids prevents its application in TTNT.

Chapter 3

A Distributed, *Ad Hoc* Joining Algorithm

Conventional PKI is cumbersome in dealing with unknown, *ad hoc* connecting entities because most protocols involve frequent, trusted, third party access. Due to the nature of the TTNT environment, users require secure and rapid connections. However, users may not be close enough, either in a physical or networked sense, to access a trusted third party. In this chapter, we discuss our algorithm for key creation, management, and exchange that will provide *ad hoc* authentication without use of a third-party.

3.1 PKI Assumption

A traditional PKI implementation is not suitable for TTNT because of the complications with the wireless and mobile aspects. Typical PKIs rely heavily on constant trusted third party access, contrary to a mobile, *ad hoc* realm where there is neither the means nor the computational power to continually maintain a secure connection with such an entity. However, a PKI can serve as a valuable framework upon which to build due to the intrinsic structure embedded in TTNT's native military environment. For example, the PKI must apply a distributed authentication technique and not require trusted third-party references at the time a join request occurs, since such a restriction would inhibit our desired *ad hoc*

connectivity. We assume that trusted third parties exist in the TTNT network that are both cryptographically and physically secure. For the application of tactical aircraft, military air bases and aircraft carriers are prime candidates to act as trusted parties. These secure servers act as the lowest layer in our certification authorities (CA) infrastructure, to provide a reliable digital signature framework. While the network users are in its presence before they engage in *ad hoc* operations, the CA will perform certificate and key management to ensure that the current network state and certificates are updated. The PKI framework facilitates faster authentication methods because communicating entities will most likely be either local or inter-domain.

3.2 Certification Authorities

A certificate authority is responsible for verifying the identity of entities, issuing public and private key pairs. When issuing new public keys, a CA appends its digital signature with the public key to bind the entity's identity of the key's owner to the key pair. CAs play an essential role in a PKI because the CA structure establishes a chain of verification and trust to each user. Fig. 3.1 depicts an example of our certificate tree. In this example, CA_{00} is the root CA. It authorizes its immediate subordinate CAs with its digital signature. The lowest CAs in this example are CA_{30}, CA_{31}, CA_{32}, and CA_{33}. As shown in this figure, the fourth layer are the active TTNT participants. In our scenarios, these entities are represented as tactical aircraft. The *ad hoc* joining procedure we discuss involves two aircraft from this level. We refer to the fourth layer participants as Joining Entities (JE), and for explanatory purposes, we associate a modern tactical aircraft with each JE. We represent both JE_{40} and JE_{41} as US Navy F/A-18 Hornets, JE_{42} as a US Navy S-3 Viking, JE_{43} as a US Air Force A-10 Thunderbolt II, and JE_{44} as a British Eurofighter.

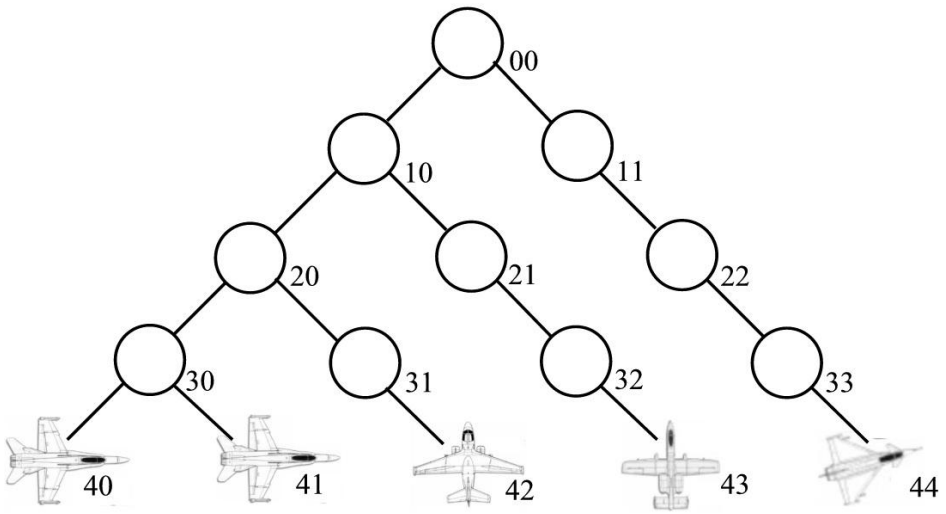

Figure 3.1: TTNT Certification Authority Structure

3.3 PKI Domain Sets

A graphical representation of the TTNT domain classification is shown in Fig. 3.2. Although network entities will form *ad hoc* connections most often with their local domain members, TTNT entities will also have membership in larger domain levels. Authentication outside a local domain is possible with a hierarchical certificate structure. The military environment, for which TTNT is specifically designed, is well suited for such a PKI.

In the TTNT environment, aircraft communicate most frequently with other aircraft from their squadron, therefore they all share a local CA. Once the verification process extends beyond the local reference, both authentication time and computational complexity increase. The second and third domains in Fig. 3.2 represent broader unit classifications in the TTNT domain scope. For example, two entities from different local domains would have to use their shared second domain CA for authentication.

In a possible TTNT implementation, the local domain level might consist of platforms from the same ship, or aircraft carrier. As show in Fig. 3.2, a sample local domain can contain tactical fighters that may exhibit different NCW characteristics. A tactical fighter attempting to join a network that is composed of fighters from his own squadron is considered

local to the network since they would all share the same immediate CA, in this case the

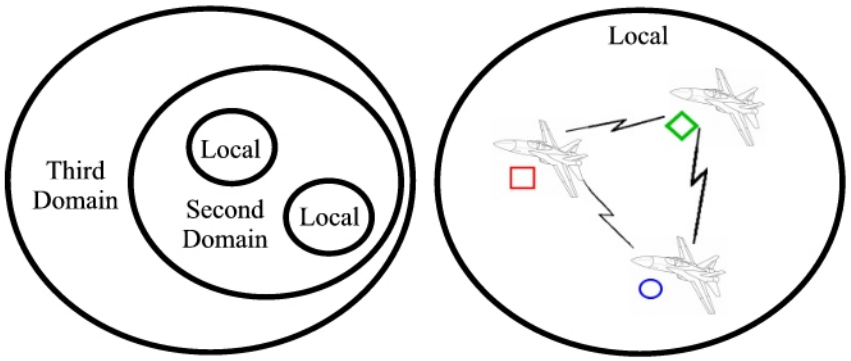

Figure 3.2: TTNT Domain Classification

aircraft carrier. Classifying TTNT entities as being from the same service would be the next highest domain. The same-service classification adds complexity to the joining process due to the increase in the number of certificates used. In a similar manner, the classifications can be further extracted to include same-country domains, and coalition domains.

3.4 Digital Certificate Specification

In an effort to explore non-standard certifying mechanisms, we generated two approaches for certificate issuing procedures resulting in certificate chains. The essential difference in these two approaches lies in the specific data that is received with the certification authority's signature.

Our initial approach had the performance advantage of allowing any common certifying authority's key held by the decision maker to immediately decrypt the joiner's key. The key could be authenticated after determining the common key, rather than having to unravel a chain of signatures down the tree from a common key. Fig. 3.3 depicts a certificate chain with four certificates. This approach has the disadvantage of requiring that all authorities in the entity's chain of command sign each platform's key. This centralized coordination is not acceptable in a distributed environment, as it would require that each level of certification

authority be provided a copy of the key to sign. Also, each CA would be tasked with verifying the identity of every key holder. This is much more effectively done at the level of authority closest to the entity itself because there are less certificates to validate.

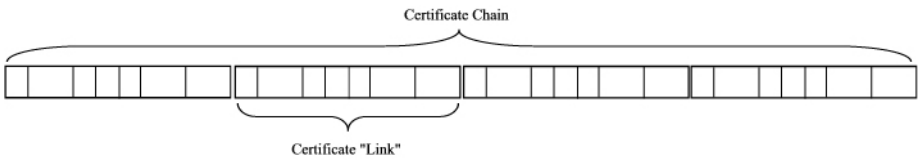

Figure 3.3: The Certificate Chain

The second approach requires that each key be signed only by its immediate certification authority. For this discussion, let A represent JE_{40} from Fig. 3.1 and B represent JE_{41}. The certificate chain that B holds contains a series of public keys, each accompanied by a data structure containing administrative information about the certification authority, all encrypted by a superior level CA. When one of B's public keys is able to decrypt a link in the chain, the result is the public key needed to decrypt the next link. Using this process, once B successfully decrypts the message, it obtains A's public key. The data structure that is appended to the message contains attributes and credentials about the CA that encrypted the message. The first entry in the data structure is a character string that will be regularly changed by the CA. Since the data structure is appended in plain text, the potential exists for cryptanalysis of the plaintext information that is encrypted directly with a private key. However, if the first entry of the data structure is dynamic, there will be less discernable difference between the end of the public key and the start of the plaintext data, making cryptanalysis more difficult. The portions of the certificate "link" that are encrypted with the symmetric key and other details of the certificate are shown in 3.4

A modification to the second approach which reduces the threat of a chosen plaintext attack is to have the previous CA encrypt a symmetric session key with its private key. This approach also reduces the threat of chosen plain text cryptanalysis because the random symmetric key is the only information encrypted directly by the issuer's private key. Like the dynamic data structure entry, the CA can mandate a symmetric key change to also minimize

a potential security risk.

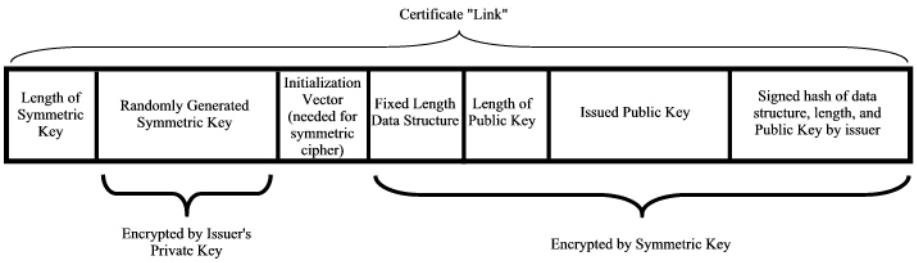

Figure 3.4: Detailed Certificate

3.5 *Ad Hoc* PKI Connection

After certificate initialization, entities are prepared to participate in the TTNT network. Table 3.1 describes the authentication algorithm in detail. The algorithm follows a Diffie-Hellman Key Exchange approach [10], [21]. As shown in Table 3.1, each Diffie-Hellman Key is created at the beginning of the protocol and successful completion of the Diffie-Hellman exchange results in each participant generating the same symmetric key, which they will use to encrypt their following communication.

Although both parties now have the same symmetric key after participating in a Diffie Hellman Exchange, there is no assurance to the server (or the client) of the identity of the communicating party. This is due to the potential for a man-in-the-middle attack in the Diffie-Hellman Key Exchange protocol [21]. Schneier suggests additional steps to refute this vulnerability through the addition of a digital signature verification [21]. In our algorithm, we include an additional procedure to prevent this threat. Since every TTNT node has a certificate chain that includes information about its public key, and the public keys of its verifying CAs, there is enough information available in the certificate chain to provide authentication.

Authentication is possible following the assumption that every TTNT participant shares at least the root CA in common with any other participant. This assumption is reasonable in a military environment where there is a chain of command to maintain accountability.

We stress however, that verification involves the root CA only in a minority of the cases. Therefore, when the server receives the client's certificate chain, it can trace the chain back to a common CA. Once the server has located the common CA, it can begin to unlock lower CAs using the wrapped public key in the certificate, as described in Section 3.4.

At this point, authentication is still not established. As shown in Table 3.1, the next step in the algorithm is for the server to verify the client's digital signature on the certificate chain. This verification is the key element in the authentication. A valid digital signature provides a degree of authentication and message integrity [21]. The authentication aspect proves that the signing private key corresponds to the public key recently obtained in the certificate chain. The message integrity proves that the certificate chain is not altered during transmission. Therefore, with a successful verification of the final digital signature, the client's identity has been proven to the server; the server has assurance that the public key it received from the certificate chain does indeed belong to the client.

Server (Network)	Client (Joining Aircraft)
Generate Diffie-Hellman Key Pair	Generate Diffie-Hellman Key Pair
Listen on Port	Connect to Port (via Jini)
Send own DH Public Key	Receive Server's Public Key
Receive Client's Public Key	Send own DH Public Key
Perform Key Agreement	Perform Key Agreement
Create Initialization Vector	
Send IV	
	Receive IV
Generate Symmetric Key	Generate Symmetric Key
Create Cipher Stream	Create Cipher Stream
	Sign Certificate Chain
Receive Signed Certificate Chain	Send Signed Certificate Chain
Extract Client's Key from Cert. Chain	
Verify Client's Signature on Chain	
Encrypt Network Session Key	
Send Encrypted Network Session Key	Receive Encrypted Key
	Decrypt Network Key

Table 3.1: TTNT Authentication Algorithm

Chapter 4

Prototype Implementation

The prototype implementation focuses on the areas of modeling, certification authority design, and improving the simulation interface. In Section 4.1 we discuss the impact of defining the relationship between TTNT entities, and present a model for network simulation interaction. In Section 4.2 we examine the incorporation of a certification authority into our simulation. In Section 4.3 we present the prototype simulation interface, followed by Section 4.4, which contains concluding remarks and observations concerning our prototype creation.

4.1 Relationship and Class Design

After the initial experimentation, we decided to approach further prototype simulations with a higher level of software engineering in the software development process. Fig. 4.1 represents a Unified Modeling Language (UML) [19] Class Diagram for the main components of our software system design. The UML is a software industry standard for software modeling and design. We use the UML class diagram in Fig. 4.1 to show the relationships between classes in our modeled system. Defining the relationships in the simulation is critical to the experiment procedures because a highly cohesive modeling with low coupling among components will aid in future scalability. The fundamental design aspect of the TTNT prototype simulation lies with how the three types of NCW entities interact. Since the underlying concept of NCW is to have adaptive roles, we needed to design the simulation to

support this property. The initial proposal consisted of a solution in which a shooter could inherit properties from a sensor, and then if it needed to, also inherit from a decision maker class. However, we use Java in the implementation, and Java does not support multiple inheritance [4]. Therefore, we constructed the design in Fig. 4.1, which illustrates that all three NCW entities inherit from the same, abstract class, `AbstractEntity`. Use of a common superclass means that a sensor, a shooter, and a decision maker will all have a common interface for interacting with other components. Even though the three entities have similar interfaces, there are characteristics that are unique to their specific datatype, which they were able to individually support in their particular subclasses. The abstract entity design choice aided in laying a foundation for future experiments.

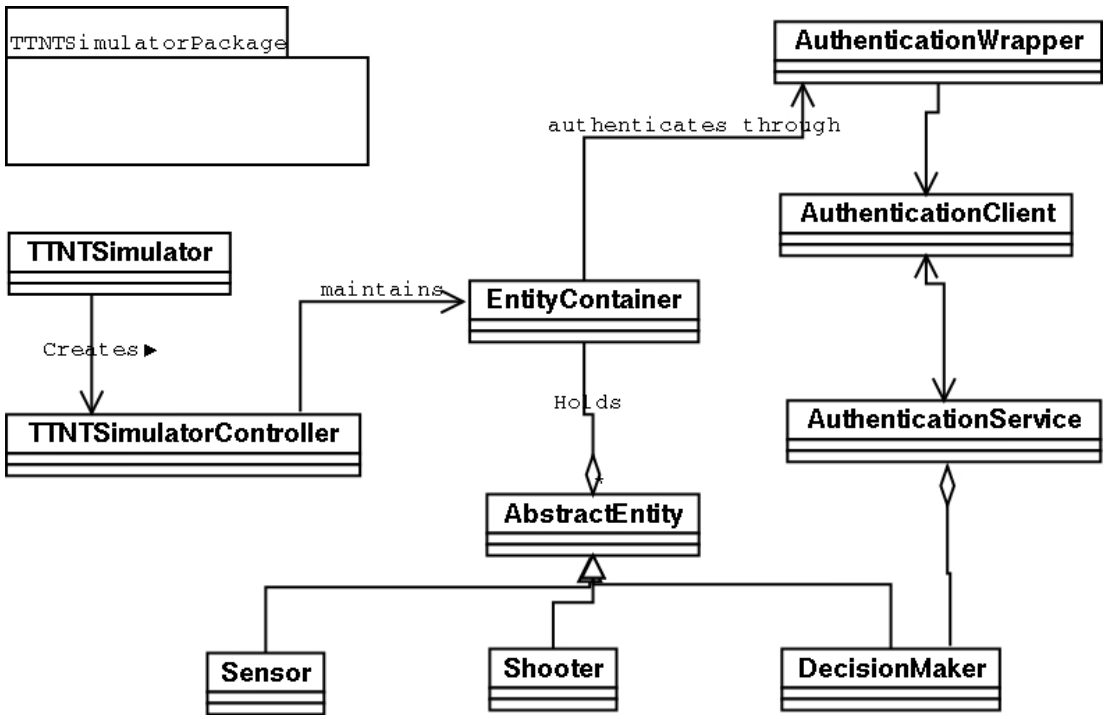

Figure 4.1: TTNT Prototype Simulation Class Diagram

4.2 Incorporation of a Certification Authority Into the Simulation

Once a TTNT entity has been issued a key, it may then participate in an *ad hoc* PKI connection. The CA only distributes keys during pre-flight as a pre-condition for an entity to join an *ad hoc* network. When entities attempt to join, we assume that they have obtained a certificate from a CA. The certificates that are issued, and therefore the respective certificate chains, are thus scalable to represent increasingly complex PKI connections. In this project, we present a PKI with four layers, although in a different implementation the number of layers may vary. A goal our experimentation presented in Chapter 5, is to measure the joining algorithm's performance in a net-centric environment. Although the hierarchical PKI is necessary to obtain net-centric behavior through organized planning, decentralized operation, our experiments focus on algorithm's ability to operate in an *ad hoc* networking environment.

In a probable application of TTNT, the CA would reside on an aircraft carrier or at an airfield's headquarters. These platforms are better suited to house a CA which requires greater computational ability to create and issue mathematically complex public key certificates. When creating a public key, the CA must work with extremely large prime numbers to proceed with the public key algorithms. These calculations are complex and require substantial computation time. Therefore, a fixed site such as an aircraft carrier would make an excellent candidate for a local TTNT CA, because of the availability of dedicated servers to perform the necessary certificate management. Once the carrier performs the initial key distribution, the aircraft may engage in the *ad hoc* joining process since they are now equipped with the necessary prerequisites for the authentication algorithm.

The CA simulation portion is modeled using an open-source cryptographic library which allows use of popular cryptographic algorithms [13]. In accordance with its license agreement, the actual implementation uses pre-defined software packages that allow user-defined key lengths for cryptographic algorithms.

4.3 Simulation Interface Design

We applied the entity relationship analysis and the class diagram in 4.1 to construct the simulation and its interface. As shown in Fig. 4.2, upon initiation of the simulation the user is presented with a message window indicating that the network is being created, and a screen that shows the simulation log. The simulation log window serves as a means to validate the inner workings of the simulation, and also may be used as a debugging aid for scenario development. As indicated in figure 4.2, the simulation created an instance of an `EntityContainer` class, which will hold only one TTNT entity in this simulation.

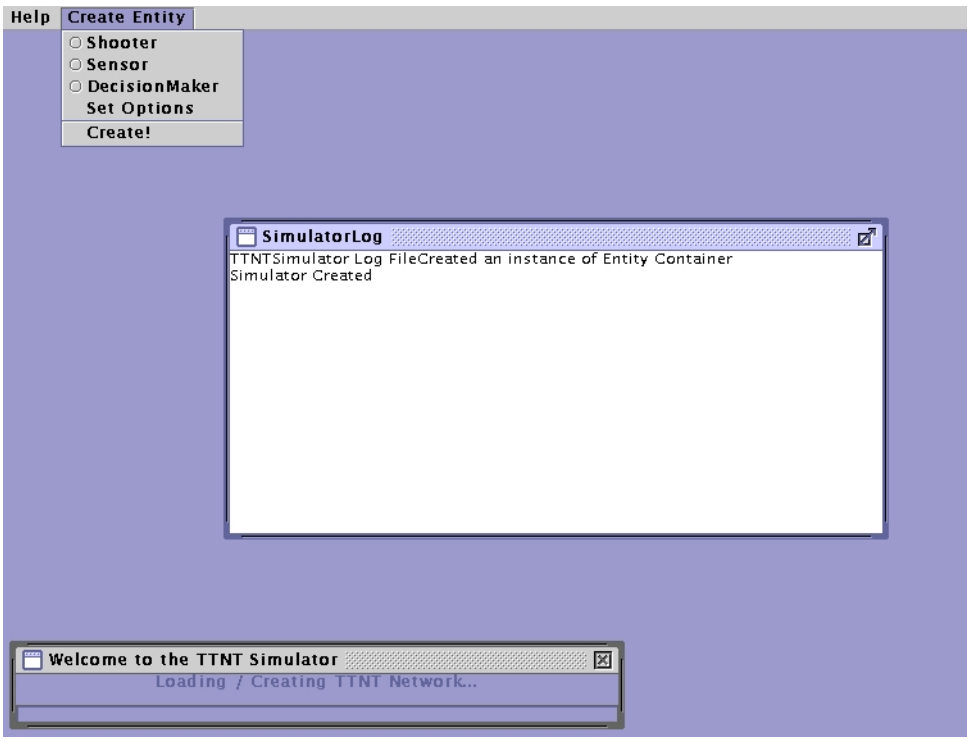

Figure 4.2: Initial Interface Screen

In accordance with the scenario, the user attempts to join the network as a sensor. The simulation notes this decision, and when `Create!` is selected, the simulation instantiates a sensor entity. Fig. 4.3 shows the screen after the simulation creates a sensor object. The sensor window indicates that a network is available to join. Additionally, the simulation log maintains itself and notes the creation of a new Sensor object.

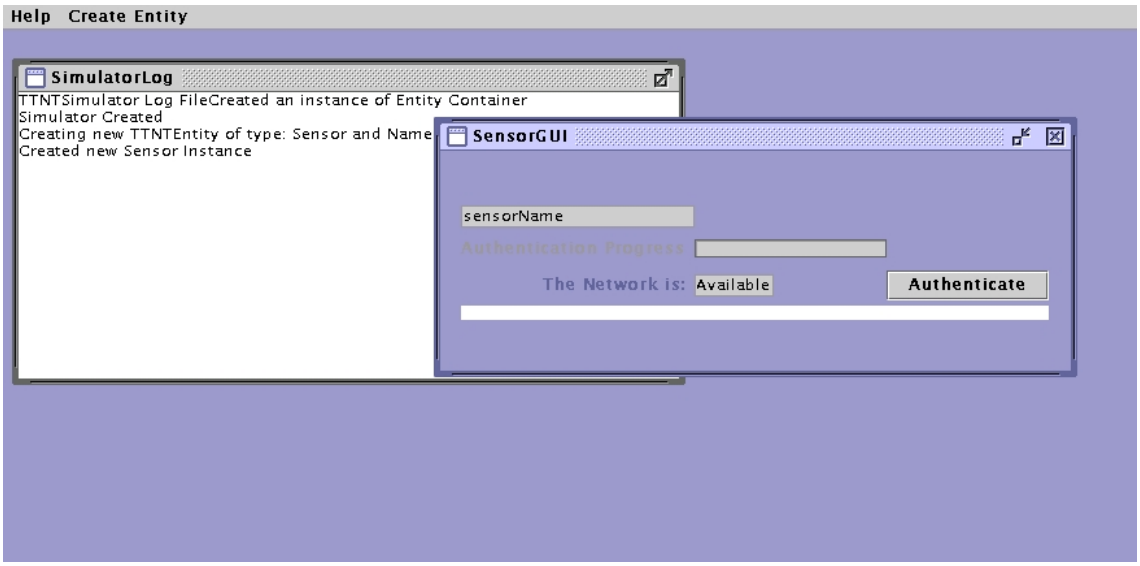

Figure 4.3: Sensor Created

Once the user presses the `authenticate` button, the simulation proceeds with the authentication process as per our algorithm. As Fig. 4.4 depicts, the steps of the algorithm appear in the simulation log which records all the network events. As noted in the simulation log, the sensor begins the authentication process via its entity container class. The platform on which the sensor resides proceeds with the authentication. After a successful authentication, the network session key is returned, allowing the sensor, and all other existing entities on that platform, to securely participate in the network.

4.4 Prototype Conclusions

This prototype provides an initial proof of concept. By defining the TTNT entity relationships, we were able to explore and propose the interaction protocols needed for sensors, shooters, and decision makers joining an *ad hoc* network without prior knowledge of the authenticating network. For example, we observed that an aircraft with many sensors, can authenticate all of its sensors at once if desired, rather than authenticate each sensor individually. This is an important finding because it emphasizes a federated, distributed en-

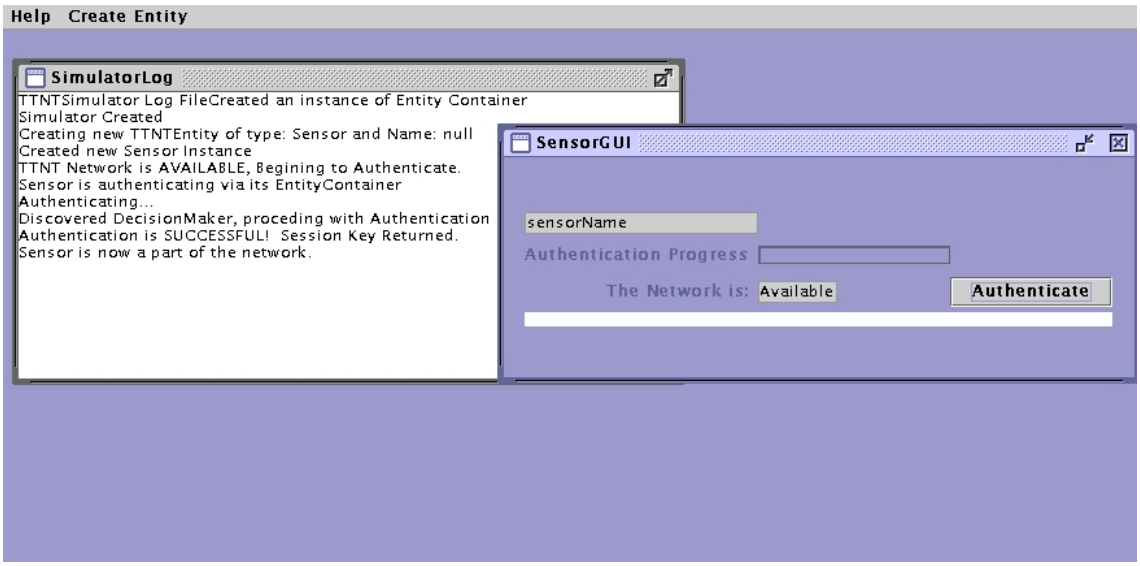

Figure 4.4: Successful Authentication

vironment and offers a method for doctrinal application. The creation of the simulation and graphical interface illustrated that the cryptographic libraries are a requirement to provide genuine authentication.

Chapter 5

Implementation Overview

We implemented our algorithm using open source software running under the GNU/Linux operating system. We used Sun Microsystems' version of Java and cryptographic libraries to provide RSA functionality. Likewise, we use Jini Networking Technology to provide the application layer of our network stack, coupled with Sun's Remote Method Invocation (RMI) for network communications. Client/server testing occurred between two machines each equipped with a Lucent Orinoco (silver) 802.11b wireless Ethernet card.

We developed two main Java packages for our algorithm development. The first, the `PKIGen` package creates the specified digital certificates, including public key generation. The second, `Authentication` package, is the source code that implements the authentication exchange between a client and server.

5.1 Motivation for Jini Implementation

To establish the desired *ad hoc* joining implementation, we model the authentication procedure as a Jini service. Since the authentication service is responsible for determining whether or not a client can be allowed to access the network, the client must first request this service before the network grants other service requests. Before a client can access the service however, the service object must register with a lookup service. As shown in Fig. 5.1a, the service uploads its proxy to the lookup service so that clients may download it. The object's

service proxy is downloaded by clients, and provides the interface for interaction with the service. When a sensor object requests authentication from the network, the lookup service responds with the downloadable proxy, as seen in Fig. 5.1b. Once the lookup server has responded, the sensor then contacts the authentication server directly to proceed. This portion of the joining algorithm is implemented in Jini by having the sensor object contact the authentication server via its published proxy, which is shown Fig. 5.1c.

5.2 Modeling the Authentication Protocol as a Jini Service

We modeled and tested the algorithm using Jini technology [11]. Jini's net-centric infrastructure delivers network plug-and-play features making Jini useful in situations where a user wants to request a service from the network, but does not know the server's location. Using Jini technology, the network provides the user with the desired service by first locating and then installing the necessary components. As a result of the advanced network management routines used by Jini, Jini provides automated synchronization of network resources, which is an important requirement of TTNT.

Two other factors motivated the decision to model the authentication protocol as a Jini service. First, a Jini service is universally available across the network to a client. In particular, a client does not need any location information about the service, since all of the discovery methods are handled by the Jini protocol. Therefore, the specific network operations appear transparent to the client. This feature facilitates the net-centric component required for TTNT operation. We use Jini's default lookup service mechanism to support the lookup routines for the authentication service. This service follows the traditional Jini model, in which the service will first publish itself with a lookup service. When a client requests the service, it probes for a lookup service and then downloads the requested service's proxy object. A proxy object is an interface that relays communication from one object to another. In our implementation, the proxy object allows the client to communicate with the

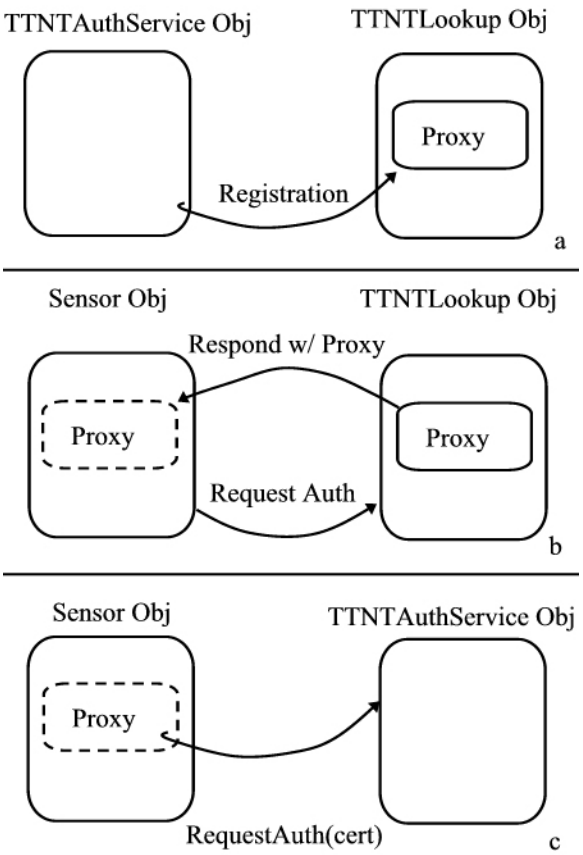

Figure 5.1: Abstract Jini Authentication Behavior

service without knowledge of the service's location. The second advantage of authentication protocol implementation as a Jini service is that the server on which the authentication service resides can maintain supervisory privileges of the algorithm operations.

5.3 Key Generation and PKI Configuration

To create the digital certificates needed for the client authentication, we modified a hybrid file encryption program that used a Rivest-Shamir-Adelman (RSA) public key pair and an Advanced Encryption Standard (AES) symmetric key [10]. The hybrid scheme is optimal for file encryption as symmetric cryptographic algorithms are generally faster than asymmetric algorithms because symmetric algorithms are equally secure with smaller key sizes. This per-

formance difference is especially noticeable when processing large blocks of data. Although the certificates we are creating are not excessively large (when encoded and written to a file, a public key is 294 bytes) our hybrid approach also serves as a barrier to cryptanalysis. The RSA private key is only used to encrypt the AES key, which in turn encrypts the rest of the message payload, thus decreasing the data encrypted by a private key. As discussed in Section 3.3, by encrypting the AES key with the RSA private key, the AES key encrypts more plaintext information which reduces the possibility for successful cryptanalysis on the RSA private key. We chose key lengths of 2048 bits for the RSA keys and 256 bits for the AES keys. The disparity in key length derives from the two different cryptographic algorithms. Since the RSA algorithm is based on factoring large prime numbers in order to ensure computational security, a RSA key is usually significantly larger than symmetric keys, like AES, that maintain the same level of security.

Modifications to the hybrid file encryption software included adding the certificate authority's data structure to the certificate. The data structure is a Java fixed byte array that contains fields which hold characteristic information about the key, the key issuer, and the owner of the key. The purpose of the data structure is to aid other TTNT participants in determining the characteristics and access policies of different keys. The characteristic field information is similar to the X.509 certificate standard which provides a complete description of different fields, and is a widely accepted standard for commercial digital certificates [21]. In our implementation, relevant characteristics are inserted into the data structure during the key creation process and are appended to the rest of the certificate.

The goal of this modified file encryption program is to produce certificates that will aid in the authentication process in a TTNT *ad hoc* join. Therefore, the CAs and corresponding certificates need to follow our four layer PKI specifications. In order to establish authentic PKI criteria, we consulted numerous Internet standard Request For Comments (RFCs) that describe the security focused process of the initial CA creation. Although this is a research implementation, we considered and applied some of the security policies from a commercial PKI [2]. Specifically, we abstracted the CA and key creation code and procedure from the authentication implementation into a separate Java package. This abstraction also empha-

sizes that efficient and secure *ad hoc* connectivity can occur in a net centric environment through a well organized PKI. During the *ad hoc* authentication process, the joining algorithm relies on the trust association given to the CAs and JEs, during the initialization of the PKI, to instantly establish the existence (or lack) of a trust relationship between two TTNT entities.

5.4 The Authentication Procedure

We created a Java package, `Authentication`, that detailed the specific algorithm procedures. The code in this package uses a cryptographic library and Jini networking technology to implement the authentication protocol. As is typical with Jini service design, there are several iterations of interfaces to facilitate ease of modification at each level [7]. Table 5.1 describes the files in the Authentication package. AuthenticationClient.java and AuthenticationService.java are the two files that contain the main implementation of the client and server side algorithm.

File	Description
`AuthenticationClient.java`	Contains the main logic for the client-side implementation
`AuthenticationService.java`	Contains the main logic for the server-side implementation
`AuthenticationServiceProxy.java`	Forwards the methods calls from the Client Interface to the Network (RMI) Interface
`AuthenticationServiceWrapper.java`	Contains the necessary Jini functions to register the service with the network
`CADataStructure.java`	Specifies the field descriptors in a TTNT certificate
`ClientAuthenticationInterface.java`	Allows the client implementation to communicate with the Jini Service Object
`DiffieHellmanExchange.java`	Provides the ability for a one-time Diffie Hellman Key Exchange
`RMIAuthenticationInterface.java`	Allows the server implementation to communicate with the Jini service object
`Utility.java`	Various iterative tasks like file operations, decryption and encryption capabilities

Table 5.1: Authentication Package File Contents

5.5 Authentication Implementation Details

When the server is created, it immediately pre-processes all of the computationally intensive operations so as not to slow down the *ad hoc* authentication. Fig. 5.2 shows the code fragment that initializes the server. The first operation performed is the creation of an initially empty stack data structure which will eventually hold the certificates from the client's certificate chain. The first operation in this code is the generation of the symmetric network session key. If a successful authentication occurs, the client receives a copy of the key, which is shared among all authenticated participants on the network. Method `loadPublicKeyRing` invoked in the next line, searches through the server's public key ring and loads into memory all the previously validated keys. Although the only public key the server needs to authenticate any other entity is the root CA's key, the possession of other trusted public keys in the key ring serves to reduce authentication time and does not cause a security issue because the server previously authenticated each key on its key ring.

```
public AuthenticationService () throws RemoteException
{
     clientCertificates = new Stack ();
     try
     {
          generateNetworkSessionKey ();
          loadPublicKeyRing ();
          System.out.println ("Authentication Service Running");
          dH = new DiffieHellmanExchange ();
     }
     catch (Exception e)
     {
          System.out.println (e.toString ());
     }
}
```

Figure 5.2: Authentication Service Initialize Code

After the initialization of the Diffie-Hellman key exchange protocol with the instantiation of a new `DiffieHellmanExchange` object, the server waits for client interaction. There is

a subtle difference here between a traditional client/server application and our net-centric empowered implementation: traditional network services wait for the client to connect directly, while the Jini service waits for the client to invoke its methods directly. Through Jini technology, a client may directly access the methods and data of the server as if it were local code. This method invocation, combined with the interface abstraction described earlier, allows specific client implementations to differ, as long as they still interface with their network mediators. In our implementation, the `ClientAuthenticationInterface` mediates between the client and the RMI protocol. If we wanted to replace RMI with a different network protocol, we need to ensure that the `AuthenticationClient` class can still properly work with the `ClientAuthenticationInterface` class. If the client implementation needs to receive an update, or a complete change, the software can be easily upgraded without disturbing the network performance.

After the server is running and waiting for a connection, the client initiates the rest of the algorithm. After the client discovers the authentication service using standard Jini lookup policies, it runs the authentication code. Fig. 5.3 shows the subroutine that outlines the main client authentication logic. This authentication method is called from the `AuthenticationClient` class' main method, and the Jini service object is passed to this function. The service object is modeled as a `ClientAuthenticationInterface` object that allows the client to interface with the network protocol. Again, this abstraction supports further expansion, and allows even the network protocol to change without impacting the client's implementation. The client initiates the Diffie-Hellman algorithm by running the `invokeClientSide` method from the `DiffieHellmanExchange` class. Both the client and the server generate the same symmetric key from the Diffie-Hellman exchange [21], which the client will use to encrypt its certificate chain when sending it to the server.

The client sends the server its entire certificate chain by invoking the server's method: `sendServerSignature`. The code fragment shown in Fig. 5.4 depicts some of the operations performed on the client's certificate chain. The server decrypts the certificate chain using the arranged Diffie Hellman symmetric key. Once the server obtains the key, it must then traverse the certificate chain, verifying each certificate to obtain the client's public key.

```
public void authenticate (ClientAuthenticationInterface servobj)
{
    try {
        dH.invokeClientSide (servobj);
        DHSessionKey = dH.getClientSessionKey ();

        //Signing and sending over client certificate chain
        signCertificateChain (servobj);

        //Receive the Encrypted Network Session Key
        System.out.println ("Receiving Encrypted Network Session  Key");
        byte[]encryptedNetworkSessionKey =
            servobj.getNetworkSessionKey ();
        decryptNetworkSessionKey (encryptedNetworkSessionKey);

    } catch (Exception e) {
            System.out.println ("Error: " + e.toString ());
    }
}
```

Figure 5.3: Authentication Client Initialize Code

The final verification occurs when the server checks the signature on the entire certificate chain. Until this point, the algorithm is vulnerable to a man-in-the-middle attack, in which a malicious user masquerades as the real client. However in our approach, the server verifies the client signature on the entire certificate chain. Therefore, even if an illegitimate user completes the algorithm as far as signing the certificate chain, when the signature is checked for validity against the public key obtained from the public key extraction, the validation will fail.

In the `extractPublicKeyMethod`, the server traverses the client's certificate chain, comparing the client's public keys to the public keys stored in the server's key ring. There are two main operations in this method: the first determines if the server has any of the public keys associated with the client's certificates in the certificate chain, the second focuses on verifying each certificate in the client's certificate chain. The `while` loop in Fig. 5.5 goes through the process of comparing each one of the client's certificates stored in the Stack data structure, `clientCertificates`, to those keys stored in its own public key ring.

```
public void sendServerSignature (byte[] encryptedClientSignature)
throws RemoteException
{
     ...
     extractPublicKey ();
     clientPublicRSAKey = Utility.loadRSAPublicKeyFromFile
          (KEY_RING_PATH + clientPublicKeyFile);
     Signature signature = Signature.getInstance
          ("MD5WithRSA");
     signature.initVerify (clientPublicRSAKey);
     signature.update (certChainWithTimeStamp);

     boolean authorized = false;

     try {
          authorized = signature.verify (clientSignature);
     } catch (SignatureException se)
     {
         System.out.println ("Invalid Padding ");
     }

     if (authorized)
     {
         System.out.println ("Client signature matches");
     }

     else
     {    System.out.println ("Client signature is INVALID,
               suspect Man-in-the-Middle ATTACK!");
          System.out.println ("PROGRAM WILL NOW TERMINATE");
          System.exit (0);
     }
}
```

Figure 5.4: Signature Verification Code

Our actual implementation improved our prototype version by the addition of a legitimate cryptographic library. This has allowed us to obtain a more accurate simulation and enabled us to produce an implementation with full secure functionality. We designed the software to allow us the flexibility in modeling different TTNT scenarios and to test our PKI domain sets.

```
private void extractPublicKey () throws Exception
{
    System.out.println ("Beginning to extract Client's Public Key");
    boolean haveCertificate = false;
    String tempCertificateName = new String ("null");
    String clientEncryptedPublicKeyFileName = new String ("null");
    Stack reverseClientCertificates = new Stack();

    while (haveCertificate == false)
    {
        tempCertificateName = String.valueOf (
             clientCertificates.pop ());
        reverseClientCertificates.push(tempCertificateName);
        if (clientEncryptedPublicKeyFileName.equals("null") )
             clientEncryptedPublicKeyFileName = tempCertificateName;
        System.out.println ("\tThe current value of tempCertificateName
             is: " + tempCertificateName);
        haveCertificate = searchKeyRing (tempCertificateName);
    }
    traverseChain(reverseClientCertificates);
    return;
}
```

Figure 5.5: Certificate Extraction Code

Chapter 6

Experiments

We conducted the experimental implementation of the joining algorithm using Sun Microsystems' Jini Technology version 1.2. In Section 6.1 we describe the procedures of the algorithm as applied to joining the *ad hoc* network. In Section 6.2 we illustrate a potential TTNT scenario application. In Section 6.3 we examine the assumptions specific to the experimental scenario. In Section 6.4 we outline the extended experiments and we present our hypothesis for the results.

6.1 Joining Phases

Our joining algorithm is divided into three phases: initialization, discovery, and authentication. As shown in Fig. 6.1, the initialization phase requires that the authentication service provider register itself with a decision maker entity. We assume that this phase occurs before a joining entity begins authentication. This allows the decision maker entity to provide a

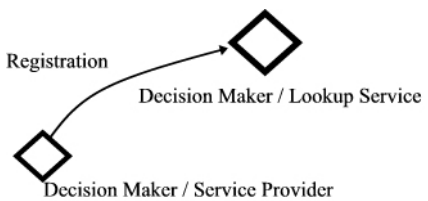

Figure 6.1: Initialization Phase of the Joining Algorithm

dynamic lookup service for services that subsequently register with the decision maker. In our scenario, a joining entity searches for and locates a TTNT network during the discovery phase. To focus our discussion of operations during this phase, we make the assumption that the network exists, and that the joining entity discovers the network.

Fig. 6.2 depicts the discovery phase and illustrates a joining entity attempting to locate a network. The actual discovery protocol is independent of the joining algorithm. A multicast protocol is preferable due to the wireless nature of TTNT, and we currently use Jini's multicast protocol [11] for our implementation.

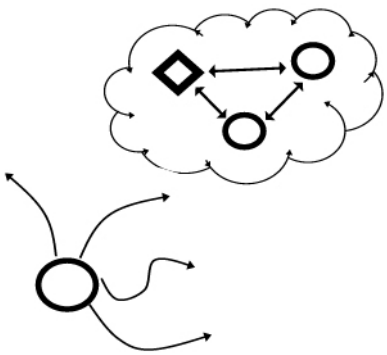

Figure 6.2: A Multicast TTNT Discovery

In order to authenticate, the joining entities participate in the multi-part procedure, which was described in detail in Section 3.5. Here we present the authentication process from the clients perspective: obtain a proxy; participate in a Diffie-Hellman key exchange; send a certificate chain to an authentication server; and await authentication service acknowledgment before the network finally accepts the client.

6.1.1 Obtain Proxy

As shown in Fig. 6.3a, the first procedure in the authentication phase is for the lookup service provider (LS) to provide a proxy to the joining entity (JE) so that it may communicate directly with the authentication service provider (AS). We stress that the JE does not need prior knowledge about the location or existence of the AS, it only has to find the LS via a

multicast discovery protocol.

6.1.2 Participate in Diffie-Hellman Key Exchange

The client calculates a Diffie-Hellman key pair, sends the server its public key, and then generates a symmetric session key to encrypt its certificate chain. Our incorporation of the Diffie-Hellman key exchange protocol into our algorithm was described in Section 3.5.

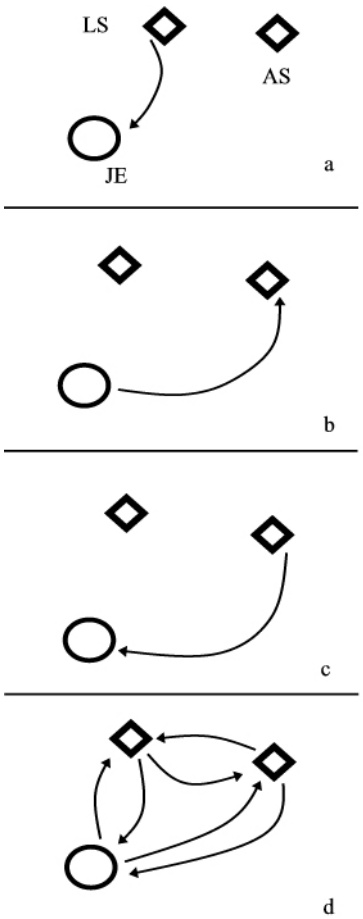

Figure 6.3: TTNT Authentication Phase Overview

6.1.3 Client Sends Certificate

Once the steps shown in Fig. 6.3a-c have completed, and both the client and the server have generated the same symmetric key, the client encrypts its certificate chain and sends

it directly to the authentication service, which is depicted in Fig. 6.3b. The authentication service then extracts the client's public key, computes the validity of the client's digital signature, and decides whether or not to authenticate the entity based on the signatures validity.

6.1.4 Authentication Service Acknowledgment

As shown in Fig. 6.3c, the authentication service responds directly back to the joining entity with an acknowledgment. The client sends pertinent routing information to the service when sending its certificate chain, ensuring that the service is able to respond to the client directly. If the acknowledgment is positive, the server encrypts the network symmetric session key with the client's public key, obtained from the certificate chain, and sends it to the client, otherwise the client is not authenticated.

6.1.5 Acceptance

Assuming that the entity was positively authenticated, the client may now participate in the TTNT network, as shown in Fig. 6.3d. Since the client now possesses the network session key, it may send and encrypt data over the network, and only authenticated participants are able to correctly decrypt the traffic. Now that the client is authenticated, it may also serve a role in authenticating other entities desiring to join the network.

6.2 Reference Scenario

In Fig. 6.4, the authentication procedure discussion from Section 6.1 is applied to a prototypical TTNT scenario. In this scenario, a sensor on the aircraft is trying to join a pre-established network of entities from similar aircraft. The sensor may be the aircraft's radar that has discovered a target and it is attempting to communicate to neighboring aircraft that a target exists. The joining aircraft initiates the algorithm. Once the network has heard the request, the network (via the lookup service provider) responds with the location of the authentica-

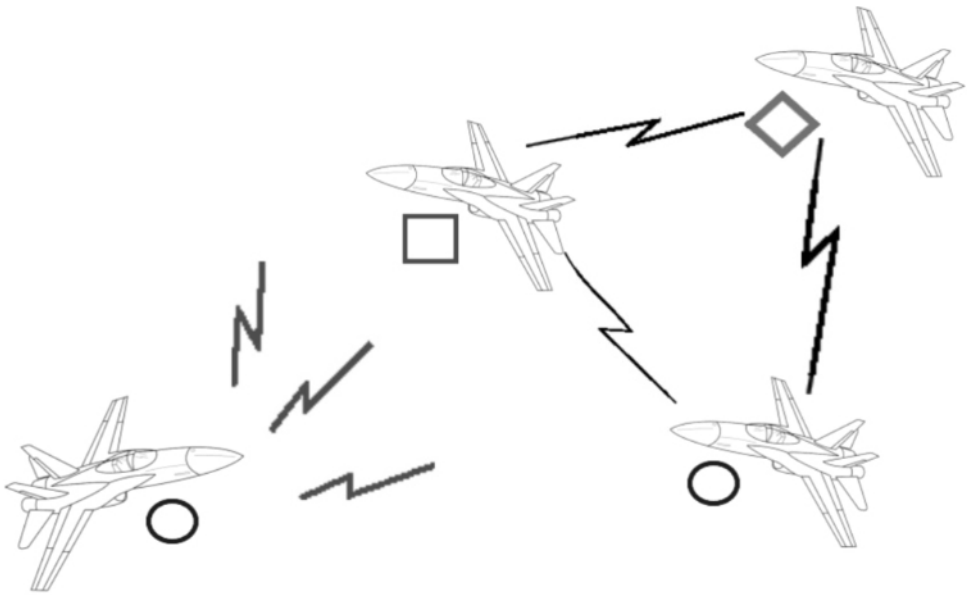

Figure 6.4: TTNT Scenario Application

tion service provider, which as shown in Fig. 6.4, resides on the decision maker (diamond) aircraft. The joining aircraft then contacts the authentication service directly and sends its certificate. If the authentication is successful, the authentication service responds with a session key. The session key provides the joiner access to the network and its services. The services in this case might be weapons on another aircraft that are appropriate to engage the target.

6.3 Reference Scenario Assumptions

Our experimentation encompasses the modeling of different TTNT scenarios using the Jini infrastructure. We consider TTNT entities in the reference scenario when one entity is requesting to join an established network, and the node it is communicating with shares a local CA. This is a situation in which an *ad hoc* connection is formed, and the joining algorithm performs at its fastest because both parties' public keys are signed by the same local CA. We assume that the two entities have a high level of trust due to their pre-established relationship. The joining algorithm scales up to handle similar scenarios when

the common CA is located outside the local group. Although the number of steps required to find the common CA increases in these cases, the entities still have a pre-defined relationship that establishes complete trust *a priori*.

6.4 Extended Scenarios

To explore the scalability between local TTNT participants (a same-ship PKI domain classification) and our most complex PKI domain set, represented by coalition forces, we changed the client's certificate chain to reflect a different CA structure. The Joining Entities (JE) shown again in Fig. 6.5, which are the fourth layer of our PKI, are the main actors in the experimentation. In each experiment, JE_{40} acts as the authentication server. Therefore, there are four different experiments, representing the four eligible clients: JE_{41}, JE_{42}, JE_{43}, and JE_{44}.

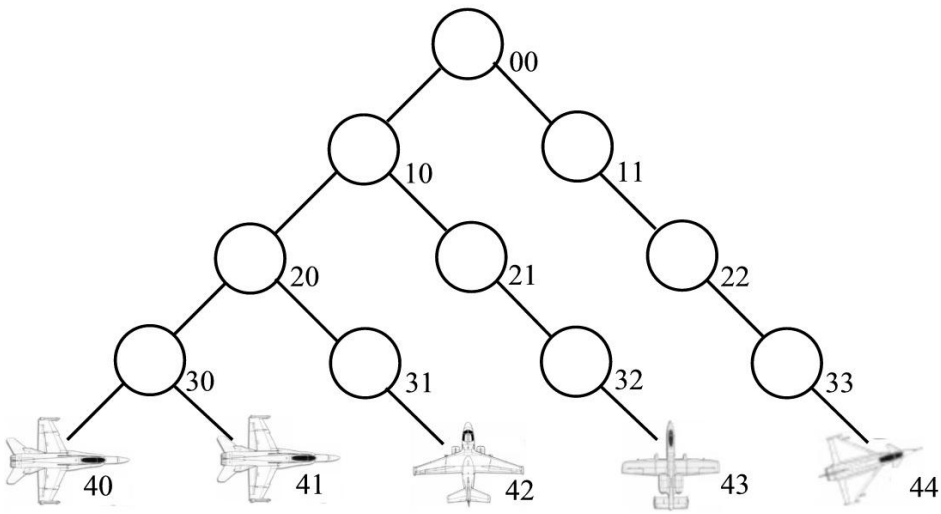

Figure 6.5: TTNT Certification Authority Structure Revisited

6.4.1 Same Ship Scenario

In this scenario, JE_{41} is attempting to connect to the server, JE_{40}. According to the PKI classification, this is a same ship environment, where, for example, there are two F/A-18

Hornets from the same aircraft carrier trying to establish a connection. In all the experiments, we represent JE_{40} as carrier-based F/A-18. We expect that this scenario will have the fastest running time since both aircraft share the same CA.

6.4.2 Same Service Scenario

In the JE_{40}-JE_{42} experiment, the two participants do not share the same local CA. The shared CA in this scenario is CA_{20}, which requires JE_{40} one additional certificate verification. We represent JE_{42} as a S-3 Viking trying to join to authenticate to the F/A-18 Hornet.

6.4.3 Joint Scenario

This experiment involves two joint aircraft, the F/A-18, and a USAF A-10, which is JE_{43}. The shared CA among these two aircrafts is now CA_{10}, which requires two additional certificate verifications. Although we have chosen to represent this scenario with a USAF aircraft, all aircraft not in the same service as the server fall into this domain classification, and exhibit the same certificate chain length.

6.4.4 Coalition Scenario

The scenario where there is the greatest certificate chain length difference from the client and the server is when JE_{44} attempts to authenticate with JE_{40}. We depict this situation as a British Eurofighter aircraft connecting with same F/A-18 described in the earlier experiments. The certificate chain difference is greatest in this scenario because the shared CA is CA_{00}, the root CA.

6.4.5 Hypothesis

We expected that there would be a strict linear progression from the same ship scenario to the coalition scenario. The only variable in these experiments is the PKI domain classification, which indicates a different certificate chain length for each experiment. For example, the

certificate chain in the coalition experiment is four times as long as the same ship service, and we predicted that each experiment would grow by a constant factor. Also, we assumed that there would be a high level of consistency within each experiment since there is no change in the certificate chain length.

Chapter 7

Analysis of Results

The implementation supported our algorithm's design goals of allowing distributed, *ad hoc*, and net-centric connections. The location and procedures of the authentication service are transparent to client. The client accesses the service through a net-centric lookup method, allowing the client to form an *ad hoc* connection. The client connects to the service without knowledge of the service's location, and without knowledge that the service indeed existed.

7.1 Measurement Decision

Our initial goal for the experimentation was to measure the clock time (in milliseconds) for our algorithm's authentication of a client in each of the four scenarios: same ship, same service, joint, and coalition. We determined that the authentication time was best measured as the time required to proceed through the algorithm, treating all necessary key generation as pre-processing artifacts external to the performance of our algorithm. We assume that before a client/server pair begins authentication they will pre-generate the session keys to prevent unnecessary delays in the actual procedure. The measurements are recorded by an `AuthenticationClient` object as shown in Fig. 7.1.

To facilitate the collection of our measurements, a Java `Date` object, `runTime`, is initialized twice to obtained the time immediately before authentication, and the time directly afterwords. The `authenticate` method gives the details for our authentication process and

```
Date runTime = new Date ();
startTime = runTime.getTime ();
System.out.println ("Start time is: " + startTime);

cl.authenticate (servobj);

runTime = new Date ();
endTime = runTime.getTime ();
```

Figure 7.1: Authentication Measure Probe

is located in the `AuthenticationClient.java` file, shown in Fig. 7.1.

7.2 Initial Results

We ran the algorithm in each scenario six times, and each time collected the time required for the authentication process for all four scenarios as shown in Table 7.1. Due to the considerable amount of time necessary to initiate the necessary software for the experiments, and then reset the conditions for each trial, we only had time to collect six measurements per client in accordance with our experiment timeline. In Table 7.1, the bottom row gives the mean of all experiment scenarios of a specific type. Upon analysis, the results clearly indicate that the increase in time from the first scenario to the last is not linear, and did not conform to our initial projections.

41	42	43	44
7407	8454	7052	7896
8516	7477	8933	17131
8411	9935	8389	9751
11607	6729	7881	9341
7866	6718	9173	10925
10724	7045	7078	10564
9089	8136	8084	10935

Table 7.1: Total Authentication Running Time (ms)

7.3 Investigation of Experimentation Environment

The non-linear growth shown in Table 7.1 contradicts our initial hypothesis of a steadily increasing running time. To discern the reason for this unanticipated behavior, we investigated two major environmental factors. The first investigation was to ensure that the Java run-time environment was acting as expected. The results from the this test are discussed in Section 7.3.1, and indicate that the Java Remote Method Invocation Daemon (`rmid`) was causing unexpected delays involving threads created by the Java Virtual Machine that were unrelated to our simulation. As a result we altered our measurement indication to obtain more meaningful data. The second area we investigated was the time required to perform only the certificate extraction in our algorithm, which is discussed in Section 7.3.2.

7.3.1 Java Run Time Benchmarking

To explore the possible causes for the unanticipated results we obtained from our simulation, we challenged our assumption that the Java run-time environment is computationally consistent. The Java programming language runs via a Java Virtual Machine (JVM) for program execution. Any machine with a JVM, regardless of the operating system or platform specifications, is able to run any Java application. In order for a Java application to run, there must be a JVM in the Java run-time environment (JRE). If the JRE is not a stable computing environment, for example if a Java application continuously runs and consumes system resources, a Java application's performance may suffer. To test the JRE, we wrote a simple Java application based on a standard benchmark that performs a mathematical operation, $\left(\sqrt{\sqrt{10}+\sqrt{10}}\right)$ and assigns its result to a temporary value, as shown in the Java source code in Fig. 7.2. We perform this operation two million times, and record the total time required for the entire two million assignments. Further, in this program we repeat each of the two million assignments two hundred times so that we can make a substantial measurement from a runtime perspective.

The results of this benchmark are shown in Fig. 7.3 and indicate stability in the JRE when running a Java application. The running time remains within a two millisecond band

```
import java.io.*;
import java.math.*;
import java.util.Date;
public class comparisons
{   public static void main( String args[])
    {   Date startTime;
        Date endTime;
        double temp = 0;
        for (int y=0; y<200; y++) {
            startTime = new Date();
            for (int x=0; x<2000000;x++) {
                temp = Math.sqrt( (Math.sqrt(10) * Math.sqrt(10) ));
            }
            endTime = new Date();
            System.out.println(y + " " + String.valueOf(
                endTime.getTime() - startTime.getTime() ));
} }    }
```

Figure 7.2: JRE Benchmarking Program: comparisons.java

throughout the entire duration of the benchmark. The most probable reason for the ten millisecond difference between the first measured time and the rest of the experimentation, shown in Table 7.1 is that there are many initialization procedures that are handled internally by Java that slow down only the first run.

The results of this initial benchmarking were inconclusive; the analysis of the timing did not offer an explanation as to the non-linear algorithmic running time. Therefore, we decided to run the benchmark again, this time including the conditions that exist while our algorithm implementation runs. Since we implemented our algorithm using Jini networking technology, there are some additional procedures that must be included at run-time to start a Jini application. To provide the ability to download and transport code, Jini applications rely on the RMI protocol [11]. Jini and Java applications may use the RMI protocol when the RMI daemon (`rmid`) is running on the system. When we re-tested our benchmarking program with `rmid` running, we noticed significant delays in time, on the order of seconds, which is shown in Fig. 7.4. We had encountered configuration problems while using `rmid` earlier in our experimentation, and even though we were able to obtain the desired functionality,

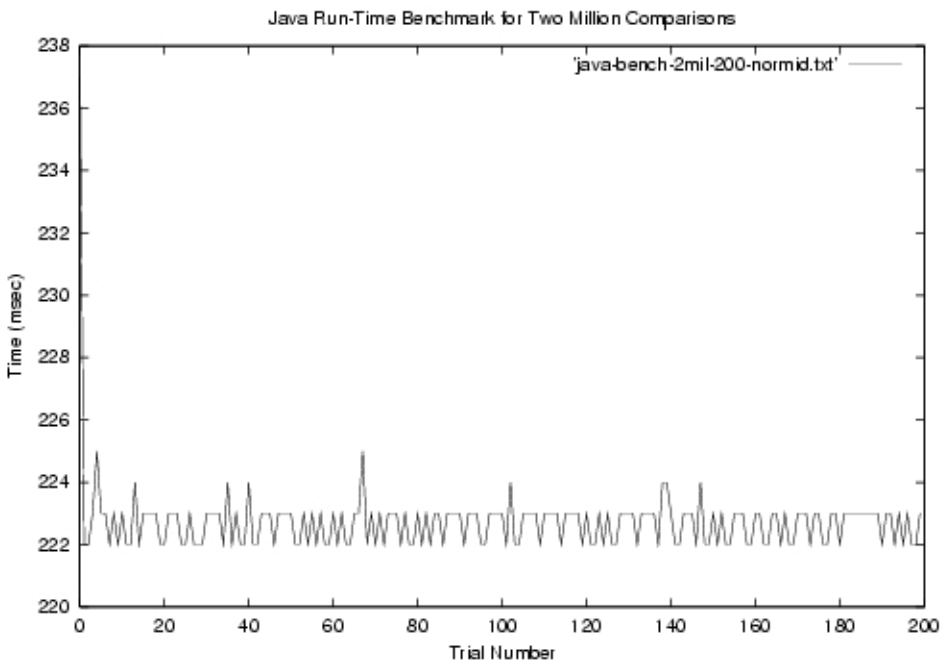

Figure 7.3: JRE Performance for Simple Operation

`rmid` continuously produced output to the screen of caught exceptions and warnings. Our analysis from the data shown in 7.4 is that the `rmid` daemon is interfering with normal Java/Jini application behavior. Since our algorithm's implementation relies heavily on `rmid` for network communication, we concluded that `rmid` impeded the algorithm's total run time and caused a disparity in the measurements we collected.

7.3.2 A More Precise Measurement Method

To obtain more meaningful statistics for our algorithm's performance in terms of scalability, we redefined the measured code sample. As discussed in Section 7.1, we originally measured the time required to complete the entire authentication procedure. However, due to the interference from `rmid`, these measurements do not accurately demonstrate our algorithm's ability to operate in a net-centric, *ad hoc* environment. Therefore, we analyzed the time required for the server to perform the certificate extraction process as this is the core computational portion of our algorithm. The certificate length is the only variable between

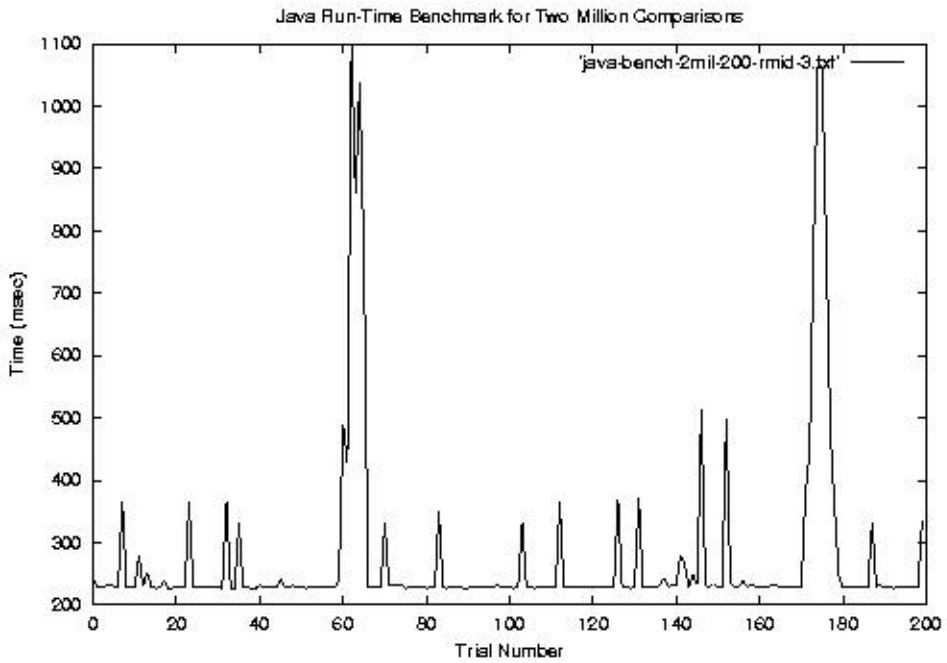

Figure 7.4: JRE Performance for Simple Operation with rmid Running

our experiment batches, and is dependent on the particular JE that is attempting to authenticate. All other factors, such as the distance between machines, the stability of the network, and the overall processor load are constant throughout the experimentation. The time sampling occurs immediately before and after the `extractPublicKey` method is invoked, as presented in Fig. 5.5. The server runs this method locally, without any network communication, and we expected that by eliminating the `rmid` portions of the algorithm, the results would illustrate the true growth of our algorithm in terms of scalability.

When we measured the time required for certificate extraction (CE), we also recorded the entire running time, as we did in Section 7.2. As expected, the CE times are more consistent and more indicative of linear growth. As we concluded, the total running time measurements do not produce any recognizable correlation due to `rmid` interference. Although `rmid` is running while the CE occurs, we concluded that the effect is reduced in these experiments because the server performs the CE without using the RMI protocol for network communication. Table 7.2 presents our subsequent experimental data and gives the arith-

41	42	43	44	41	42	43	44
362	439	470	586	7507	15382	7379	7717
344	427	508	590	7532	7349	7479	7998
359	453	491	578	7474	7148	7820	7671
371	415	486	588	7348	7280	9400	7452
Means							
359	434	489	586	7465	9290	8019.5	7709.5
Standard Deviations							
11.2	16.3	15.7	5.26	81.7	4060	940	224

Table 7.2: Certificate Extraction Time vs. Total Run Time (ms)

metic means of each experiment batch and their respective standard deviation to the mean. We concluded from the precision of the standard deviations between the total running times and the CE times that the `rmid` effect is greatest when the algorithm performs its network communications.

Fig. 7.5 is a scatter plot of the CE times, per client. Also included in this graph is the mean time of each grouping plotted with a line between experiment batches to show the linear correlation. As illustrated in the graph, there is a near linear relationship between each experiment batch.

7.4 Analysis Conclusions

We conclude from our experimental results that *ad hoc*, distributed, net-centric joining can occur. As a result of the initial experimentation, the Java benchmarking, and the revised experiments, we concluded that our joining algorithm maintains its expected theoretical performance. By our conclusions, we expect a coalition fighter to experience only marginal delays in authentication, measured as a delay of 250 ms in our implementation, when connecting to a server with the largest difference in certificate chain length. However, our algorithm does experience delays depending on the choice of implementation and network protocol. Although Jini provides automated network centric capabilities, especially in terms of lookup and discovery protocols, the added complexity for automation hinders performance

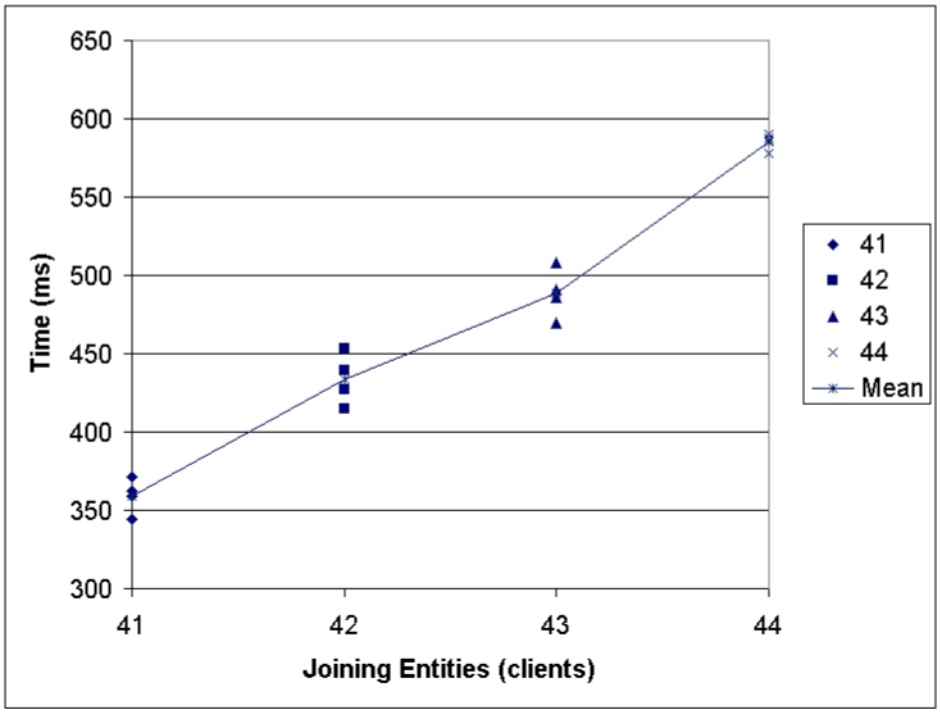

Figure 7.5: Server Certificate Extraction Time per Client

at run time.

However, in more practical terms the network should be able to accommodate a joining entity that lies outside of the TTNT domain classifications. In a typical TTNT application, this would represent a tactical fighter entity attempting to join a network composed of aircraft from a different branch of military service that may not have the required PKI certificates, or with certificates that have expired. To avoid excluding a legitimate participant under these circumstances, participating members of the network could be allowed to vouch for the identity of a entity and allow it to join. The challenging problem in this scenario is that the joining entity may need rapid access to network resources, for example information concerning a nearby target, and cannot tolerate the access time delay due to a lack of, or complex, certificate chain configuration. To facilitate a join where an entity can produce characteristics necessary for proper PKI authentication, our joining algorithm could include a trust assigning portion.

The trust assignment procedure may classify entities into different trust levels based on the security credentials they provide. For example, an entity that can produce a fully qualified certificate will be assigned to the most trusted level, whereas an entity with an expired certificate will be placed in one of the lowest levels. Included in the trust assignment process is the ability to accept recommendations from other entities. Therefore, entities already on the network may make trust assertions for a joining entity, which will reflect in the joining entity's trust classification. The security policies of TTNT will need to account for the addition of any such marginally trusted entities to maintain network security. The addition of trust classification into the TTNT joining algorithm can be effected to support rapidity of access without compromising security. This trust assigning algorithm would therefore need to adhere to the following assumptions:

- That not all entities on the network are fully trusted.

- An entity's trust rating may increase or decrease.

- A federated network of entities can perform effective trust management.

However, the trust assignment algorithm can be expected to involve only a small percentage of the joining entities, as most of the joining entities will still fall within our regular PKI classification. Furthermore, in our experiments, there is only a 250 ms delay variance between the greatest difference in the certificate chain size. If we enable trust assignment for our algorithm, in the worst case, it would only improve the run time by 250 ms, but it would not alleviate the network protocol problems. Therefore, although trust assignment may improve the overall flexibility and scalability of our joining algorithm, it will not significantly increase its performance.

Chapter 8

Conclusions

In this report we illustrated the shortcomings of the standard Jini and Bluetooth algorithms with respect to the needs of TTNT. Jini neither defines an authentication algorithm nor incorporates trust-based decisions. Bluetooth applies a symmetric key algorithm that fails to provide a sufficient level of security for TTNT.

We discussed various utility protocols that support TTNT, and described several multicast discovery protocols that could support the initial step in the joining algorithm. We also discussed the need for a distributed public key protocol to enable *ad hoc* connections in a dynamic environment.

We presented an algorithm that facilitates wireless *ad hoc* joining requests with the key assumption that an *a priori* public-key infrastructure is in place. Our experimental results support our hypothesis that our algorithm can allow successful authentication in a net-centric, *ad hoc* environment. We have also shown that the delays incurred in our algorithm are minimal compared to the delays incurred by the network implementation. Our analysis shows that we can expect the network to authenticate a F/A-18 and a British Eurofighter with minimal difference in authentication time. We demonstrated that our algorithm supports network centric warfare characteristics, specifically the ability to provide centralized management through our PKI, and decentralized operation through our *ad hoc* authentication process.

In future work, we are considering whether a separate algorithm can be created to per-

form trust management and maintenance after the entity has joined the network. This algorithm's focus would be to promote an entity's trust rating based on its behavior, and would dynamically poll the network for updated trust ratings. There is also potential for further research involving probability analysis of successful joins within the trust algorithm, especially in two scenarios: when a legitimate user is denied access, and when an illegitimate user is granted access. Also, we are considering a comparison of our algorithm with a joining algorithm that does not assume that an implemented PKI exists. In the absence of a PKI, the network would be truly distributed and decentralized, changing many assumptions concerning security.

Bibliography

[1] Alberts, David, John Garstka and Frederick Stein. *Network Centric Warfare: Developing and Leveraging Information Superiority*. (Washington: CCRP, 1999).

[2] Berge. N. UNINETT PCA Policy Statements. Network Working Group Request For Comments: 1875. `http://www.armware.dk/RFC/rfc1875.html`. (DEC95).

[3] Corson, S. and Joe Macker. *Mobile Ad Hoc Networking (MANET)*. Networking Working Group Request for Comments:2501. January 1999. `ftp://ftp.isi.edu/in-notes/rfc2501.txt` (20SEP01).

[4] Deitel, Harvery M. and Paul J. Deitel. *Java: How to Program 3rd Edition*. (Prentice Hall: New Jersey, 1999).

[5] "Digital ID Introduction." *VeriSign Digital ID Center*. 1998. `http://digitalid.verisign.com/client/help/id_intro.htm` 8JAN01.

[6] "Digital Signature Guidelines Tutorial." `http://www.abanet.org/scitech/ec/isc/dsg-tutorial.html`. *Digital Signature Guidelines*. 6JAN01.

[7] Edwards, Keith. *Jini: Example by Example*. (Prentice Hall: New Jersey, 2001).

[8] Eronen, Pasi. *Security in the Jini Networking Technology: A Decentralized Trust Management Approach*. Helsinki University of Technology Department of Computer Science and Engineering. 6 March 2001.

[9] Foster, Ian and Carl Kesselman. *The Grid: Bluepring for a New Computing Infrastructure*. (Morgan Kaufmann: California, 1999).

[10] Garms, J. and Somerfield, D. *Professional Java Security*. (Birmingham, UK: Wrox, 2001).

[11] Jini Technological Core Platform Specifications. Sun Microsystems. `www.sun.com/jini/specs/jini1.1html/discovery-spec.html#3554` (20SEP01).

[12] Klemetti, Kari and Tomi Valkeinen. "Problems with International PKI and Anonymity." `http://www.tml.hut.fi/Opinnot/Tik-110.501/1999/papers/pkiprob/pkiprob.html` *Helsinki University of Technology Department of Computer Science*. (30NOV99)

[13] Legion of the Bouncy Castle Application Programming Interface for the Java Cryptography Extension v1.12. http://www.bouncycastle.org/latest_releases.html.

[14] Macker, J. and Brian Adamson. *The Multicast Dissemination Protocol (MDP)*. Internet Engineering Task Force: INTERNET-DRAFT. http://manimac.itd.nrl.navy.mil/MDP/draft-macker-rmt-mdp-01.txt 22 October 1999.

[15] Manola, Frank and Craig Thompson. Characterizing the Agent Grid. http://www.objs.com/agility/tech-reports/990623-characterizign-the-agent-grid.html *Object Services and Consulting Inc.* June 1999.

[16] Mohapatra, Pradosh Kumar. "Public Key Cryptography." *ACM Crossroads.* 11SEP00. http://www.acm.org/crossroads/xrds7-1/crypto.html.

[17] Newman, Richard. From Up in the Sky. *U.S. News and World Report*. 25 Feburary 2002.

[18] Perkins, Charles. Ad Hoc Networking. (New York: Addison-Wesley, 2001).

[19] Schach, Stephen. *Classical and Object-Oriented Software Engineering: With UML and Java*. (New York: McGraw Hill, 1999).

[20] Schneier, Bruce. "*802.11 Security.*" Crypto-Gram Newsletter. http://www.counterpane.com/crypt-gram-0103.html#10.

[21] Schneier, Bruce. *Applied Cryptography Second Edition*, (New York: Wiley, 1996).

[22] Schneier, Bruce. *Secret and Lies: Digital Security in a Networked World*. (New York: Wiley, 2000).

[23] Security. *Security Enhancements for the Java2 SDKv1.4*. 2001. http://java.sun.com/j2se/1.4/docs/guide/security.

[24] Sirbu, Marvin and John Chung. *Distributed Authentication in Kerberos Using Public Key Cryptography*. Carnegie Mellon University. http://www.ini.cmu.edu/netbill/pubs/pkda.html.

[25] Tactical Targeting Network Technology. *DARPA*. 25NOV00. http://www.darpa.mil/baa/TTNT-ID-00.html (11JAN01)

[26] The Offical Bluetooth Website. http://www.bluetooth.com.

[27] Vainio, Juha. Bluetooth Security. http://www.niksula.cs.hut.fi/~jiitv/bluesec.html.

Appendix A

Project Timeline Reflection

In accordance with our submitted project timeline, all of the initial familiarization steps have occurred, specifically familiarization with Jini Networking Technology and the configuration and installment on a Wireless Local Area Network. Before the network setup, we conducted extensive research on existing wireless networking hardware to identify a product that was conducive to our research needs. We obtained the hardware in the early fall, and successfully configured and tested the network. Three personal computers running the Linux operating system, each equipped with an 802.11b wireless Ethernet card, make up the network. With the wireless link, the computers can communicate with each other at the relatively high speed of 11 megabits per second.

This work was based on the initial experimentation completed early in the semester, builds on an independent study class taken the previous year, SI486, and experiences over an internship I participated in involving mobile, ad hoc networking at Naval Research Labs. We were on schedule designing and classifying the relationships and structure of the simulation. The simulation was improved to support the Performance Testing stage. We performed another series of experiments and collected empirical data of the algorithm running times. This data has been analyzed to determine the effects of the algorithm's performance in a net-centric environment. We have submitted two papers to peer reviewed networking conferences. One has been accepted; one is under review.

We fell behind our proposed schedule at the beginning of the second semester due to

numerous configuration errors. Specifically, we encountered great difficulty attempting to create a certification authority (CA) using a cryptographic library. Instead of implementing a full featured CA, we decided to only create the necessary functions that a CA would provide our experiments, using more accessible packages. Another configuration problem, this time with our chosen software implementation technology, caused delays in our advanced network testing later in the semester. Through the help of numerous online support communities, we were able to alleviate the problem and perform the necessary experimentation in mid-March.

Appendix B

AuthenticationClient.java

```
/*
 * AuthenticationClient.java
 *
 * This file contains the details for a TTNT client to Authenticate
 * with a Authentication Server
 */

package Authentication;

import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceItem;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;

import java.io.*;
import java.net.*;
import java.math.*;
import java.security.*;
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import javax.crypto.interfaces.*;
import java.util.Date;

import Authentication.Utility;

//for sending encrypted messages to server
import org.bouncycastle.util.encoders.Base64;
```

```
import Authentication.DiffieHellmanExchange;

/**
 *
 * @author  joshua datko
 */
public class AuthenticationClient
{

    /** Creates a new instance of AuthenticationClient */ public
    AuthenticationClient (ClientAuthenticationInterface theClient,
    String priKeyFile) throws Exception

    {
        authInterface = theClient;
        privateKeyFile = priKeyFile;
dH = new DiffieHellmanExchange();
    }

    public void authenticate (ClientAuthenticationInterface servobj)
    {
        try
        {

            dH.invokeClientSide (servobj);
            DHSessionKey = dH.getClientSessionKey ();
            //Signing and sending over client certificate chain
            signCertificateChain (servobj);

            //Receive the Encrypted Network Session Key
            System.out.println ("Receiving Encrypted Network Session
               Key"); byte [] encryptedNetworkSessionKey =
               servobj.getNetworkSessionKey ();

            decryptNetworkSessionKey (encryptedNetworkSessionKey);


            //communicate (servobj);

        }

        catch (Exception e)
        {
```

```
        System.out.println ("Error: " + e.toString ());
    }
}

private void signCertificateChain (ClientAuthenticationInterface servobj)
{

    System.out.println ("Entering signCertificate Chain");

    String certChainFile = privateKeyFile.substring (0, 10) + ".chain";
    String path = new String ("/home/joshua/jiniscript/");
    certChainFile = path.concat (certChainFile);

    try
    {

        clientRSAPrivateKey = Utility.loadRSAPrivateKeyFromFile
           (privateKeyFile);

        //Timestamp is 8 bytes
        long timestamp = System.currentTimeMillis ();

        System.out.println ("Appending time stamp: " +
            String.valueOf (timestamp) );

        ByteArrayOutputStream baos = new ByteArrayOutputStream ();
        DataOutputStream dos = new DataOutputStream (baos);
        dos.writeLong (timestamp);

        //load the Certificate Chain from File

        System.out.println ("Loading Certificate chain from file:
            " + certChainFile);

        FileInputStream fisChain = new FileInputStream (certChainFile);

        //Read in the certificate chain
        int theByte = 0;
        while ((theByte = fisChain.read ()) != -1)
        {
            baos.write (theByte);
        }
```

```
        byte[] bytesToSign = baos.toByteArray ();
        baos.close ();

        //***************Sign the byte array
        Signature signature = Signature.getInstance ("MD5WithRSA");
        signature.initSign (clientRSAPrivateKey);
        signature.update (bytesToSign);
        byte [] signatureBytes = signature.sign ();

        //Encrypt certificate chain with Diffie-Hellman generated
            Session Key

        Cipher cipher = Cipher.getInstance ("DESede/ECB/PKCS5Padding");

        cipher.init (Cipher.ENCRYPT_MODE, DHSessionKey);


        byte[] encryptedBytesToSign = cipher.doFinal (bytesToSign);
        byte[] encryptedSignatureBytes = cipher.doFinal (signatureBytes);

        //Sending certificate chain and signature bytes to server

        System.out.println ("Sending certificate chain and bytes
            to sign to server");

        System.out.println ("\tWhich are encrypted with the
            Diffie-Hellman session key"); servobj.sendServerCertChain
        (encryptedBytesToSign); servobj.sendServerSignature
        (encryptedSignatureBytes); } catch (Exception e) {
        System.out.println (e.toString ()); }

}

/**
 * @param args the command line arguments
 */
public static void main (String args[]) throws Exception
{

    //Get the client's private key, prompt the user to enter

    BufferedReader in = new BufferedReader (new InputStreamReader
        (System.in));
```

```
System.out.println("Enter the Client you wish to use (41, 42,
    43, 44) :");

String clientString = in.readLine();

String clientPrivateKeyFile = "CA_" + clientString + "_2048.pri";

FileOutputStream log = new FileOutputStream ("logFile.txt");
        DataOutputStream out = new DataOutputStream (log);



    // Set a security manager.
    if (System.getSecurityManager () == null)
    {
        System.setSecurityManager (new RMISecurityManager ());
    }


    ServiceDiscoveryManager sdm =
    new ServiceDiscoveryManager (null,null);
    // Set up the template
    Class[] classname = new Class[]
    {ClientAuthenticationInterface.class};
    ServiceTemplate template =
    new ServiceTemplate (null,classname,null);
    // Block until a matching service is found
    System.out.println ("Looking for Authentication Service");
    ServiceItem serviceitem =
    sdm.lookup (template, null, Long.MAX_VALUE);

    // Use the Service if has been found.
    if ( serviceitem == null )
    {
        System.out.println ("Can't find service");
    } else
    {
        System.out.println ("Authentication Service Found");

        ClientAuthenticationInterface servobj =
        (ClientAuthenticationInterface) serviceitem.service;
```

```
                long startTime, endTime, algorithmRunningTime;

                long times[] = new long [10];

                int x = 0;

                System.out.println ("************************
                    ************************************************");

                System.out.println ("************************
                    ************************************************");

                System.out.println ("************************
                    ************************************************");

                System.out.println
                    ("*************************************
                    *************************************");

                // Create and then use the simple client
                //args[0] = private key file

                AuthenticationClient cl = new AuthenticationClient
                    (servobj, clientPrivateKeyFile);

Date runTime = new Date ();
startTime = runTime.getTime ();
                System.out.println ("Start time is: " + startTime);

/*
*
*/
                cl.authenticate (servobj);
/*
 *
 */
                runTime = new Date ();
                endTime = runTime.getTime ();
                System.out.println ("End time is: " + endTime);
                times[x] = endTime - startTime;
                out.writeChars (String.valueOf (times[x]) );
                out.writeChars (String.valueOf ("\n"));
                System.out.println ("This time just took: " + times[x]);
```

```
    }


}

private void decryptNetworkSessionKey (byte [] encryptedNetworkSessionKey)
{

    try
    {
        //("Create a cipher using that key to initialize it");
        Cipher rsaCipher = Cipher.getInstance ("RSA/ECB/PKCS1Padding");

        //("Read in the encrypted bytes of the session key");

        ByteArrayInputStream bais = new ByteArrayInputStream
            (encryptedNetworkSessionKey);

        byte[] encryptedKeyBytesSize = new byte[1];


        bais.read (encryptedKeyBytesSize, 0, 1);

        // ("encryptedKeyBytesSize = " + encryptedKeyBytesSize);
        //Assuming Network Session Key is of key size 256
        byte[] encryptedKeyBytes = new byte[256];
        bais.read (encryptedKeyBytes);

        //System.out.println ("Decrypt the session key bytes.");
        rsaCipher.init (Cipher.DECRYPT_MODE, clientRSAPrivateKey);
        byte[] rijndaelKeyBytes = rsaCipher.doFinal (encryptedKeyBytes);

        // (" Transform the key bytes into an actual key.");
        rijndaelKey = new SecretKeySpec (rijndaelKeyBytes, "Rijndael");

        System.out.println ("Network Session Key is: " +
            rijndaelKey.toString () );

        // Read in the Initialization Vector from the file.
        rijndaeliv = new byte[16];
        bais.read (rijndaeliv);
```

```
        rijndaelspec = new IvParameterSpec (rijndaeliv);
    }

    catch (Exception e)
    {
        System.out.println ("Exception in
            decryptNetworkSessionKey: " + e.toString () );
    }
}



private void communicate (ClientAuthenticationInterface servobj)
    throws Exception

{

    ByteArrayOutputStream output = new ByteArrayOutputStream ();
    // Create the cipher for encrypting the file itself.
    Cipher symmetricCipher = Cipher.getInstance
        ("Rijndael/CBC/PKCS5Padding");
    symmetricCipher.init (Cipher.ENCRYPT_MODE, rijndaelKey, rijndaelspec);

    CipherOutputStream cos = new CipherOutputStream
        (output, symmetricCipher);
    byte [] messageBytes;
    String message;

    BufferedReader in = new BufferedReader
        (new InputStreamReader (System.in));
    System.out.print ("Enter text to send: ");
    message = new String (in.readLine ());
    System.out.println ("The message I just read in was: " + message);

    messageBytes = Base64.decode (message);
    System.out.println ("The base64 decoded message is: " + messageBytes);
    cos.write (messageBytes);




    output.close ();
    cos.close ();
```

```
        System.out.println
            ("The supposed encrypted message is : " + output.toByteArray ());
        servobj.sendEncryptedMessage (output.toByteArray ());
        //servobj.sendEncryptedMessage(messageBytes);
    }

    ClientAuthenticationInterface authInterface;

    private String privateKeyFile;
    private PrivateKey clientRSAPrivateKey;

    private SecretKey rijndaelKey;
    private byte[] rijndaeliv;
    private IvParameterSpec rijndaelspec;

    private DiffieHellmanExchange dH;
    private SecretKey DHSessionKey;

}
```

Appendix C

AuthenticationService.java

```
/*
 * AuthenticationService.java
 *
 * Created on February 5, 2002, 6:34 PM
 */

package Authentication;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

import java.io.*;
import java.net.*;
import java.math.*;
import java.util.*;
import java.util.Stack;
import java.security.*;
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import javax.crypto.interfaces.*;
import Authentication.CADataStructure;
import Authentication.DiffieHellmanExchange;

import org.bouncycastle.util.encoders.Base64;

/**
 * This class defines the authentication server methods.
 * @author joshua datko
 */
```

```
public class AuthenticationService extends UnicastRemoteObject
    implements RMIAuthenticationInterface
{

    /** Creates a new instance of AuthenticationService */
    public AuthenticationService () throws RemoteException
    {
        clientCertificates = new Stack ();
        try
        {
            generateNetworkSessionKey ();
            loadPublicKeyRing ();
            System.out.println
                ("Authentication Service version albatross-bruce");

            dH = new DiffieHellmanExchange ();
        }

        catch (Exception e)
        {
            System.out.println (e.toString ());
        }


    }

    /** Returns the server's Diffie-Hellman public key to the client
     * @return Diffie-Hellman Public Key
     */
    public PublicKey getServerPublicKey () throws RemoteException
    {
        return dH.getServerPublicKey ();
    }

    /** Allows the client to upload his Diffie-Hellman public key
     */
    public void sendClientPublicKey
        (PublicKey clientPublicKey) throws RemoteException
    {
try
        {
            dH.setClientPublicKey (clientPublicKey);
        }
```

```
    catch(Exception e)
    {
        System.out.println (e.toString ());
    }

}

/** Sends the initilization vector to the client
 */
public byte[] getIV () throws RemoteException
{
    byte[] tempIV = dH.getDHIV ();
    DHSessionKey = dH.getServerSessionKey ();
    return dH.getDHIV ();
}

public void sendServerSignature
    (byte[] encryptedClientSignature) throws RemoteException
{
    System.out.println ("Server is running sendServerSignature");
    try
    {
//Begin a DH cipher stream,
        //will encrypt the cert chain to this stream
        Cipher cipher = Cipher.getInstance ("DESede/ECB/PKCS5Padding");
        cipher.init (Cipher.DECRYPT_MODE, DHSessionKey);
        byte [] clientSignature = cipher.doFinal
            (encryptedClientSignature);

        // Open up an output file for the output of the encryption
        String fileOutput = "client_output";
        FileOutputStream output =
        (new FileOutputStream
            (RECEIVED_CLIENT_CERTIFICATE_PATH + fileOutput));

        //Strip the Time Stamp from the Certificate Chain
        ByteArrayOutputStream baos = new ByteArrayOutputStream ();
        baos.write
            (certChainWithTimeStamp, 8, certChainWithTimeStamp.length - 8);
        byte[] certChain = baos.toByteArray ();
        baos.close ();

        output.write (certChain);
```

```
    output.close ();

    System.out.println ("Received Client's Certificate Chain");

    clientCertificates = Utility.unzipFile
        (fileOutput, RECEIVED_CLIENT_CERTIFICATE_PATH);
    extractPublicKey ();

    clientPublicRSAKey = Utility.loadRSAPublicKeyFromFile
        (KEY_RING_PATH + clientPublicKeyFile);

    Signature signature = Signature.getInstance ("MD5WithRSA");
    signature.initVerify (clientPublicRSAKey);
    signature.update (certChainWithTimeStamp);
    boolean authorized = false;
    try
    {
        authorized = signature.verify (clientSignature);
    }catch (SignatureException se)
    {
        System.out.println ("Invalid Padding ");
    }

    if (authorized)
    {
        System.out.println ("Client signature matches");
    }

    else
    {
        System.out.println
            ("Client signature is INVALID,
            suspect Man-in-the-Middle ATTACK!");
        System.out.println ("PROGRAM WILL NOW TERMINATE");
        System.exit (0);
    }

}

catch(Exception e)
{
    System.out.println (e.toString ());
```

```
        }
    }

    private void extractPublicKey () throws Exception
    {
        //Check to see if the public key already exists in the data base
        //if (searchPublicKeyDatabase(clientCertificates) == true) {
        System.out.println ("Beginning to extract Client's Public Key");
        boolean haveCertificate = false;
        String tempCertificateName = new String ("null");
        String clientEncryptedPublicKeyFileName = new String ("null");
        Stack reverseClientCertificates = new Stack();

        while (haveCertificate == false)
        {
            tempCertificateName = String.valueOf ( clientCertificates.pop () );
    reverseClientCertificates.push(tempCertificateName);
            if (clientEncryptedPublicKeyFileName.equals("null") )
                clientEncryptedPublicKeyFileName = tempCertificateName;

            System.out.println
                ("\tThe current value of tempCertificateName is :
                " + tempCertificateName);
            haveCertificate = searchKeyRing (tempCertificateName);

        }
System.out.println ("***HERE I WILL CALL TRAVERSE CHAIN***");
        traverseChain(reverseClientCertificates);

return;

    }

    private void traverseChain(Stack certificateChain) throws Exception{

String currentCertificate = String.valueOf ( certificateChain.pop());

        //Test to see if stack is empty
if (certificateChain.isEmpty() == true)
    return;

String nextCertificate = String.valueOf ( certificateChain.peek());
```

```
System.out.println ("Current Certificate is");
        System.out.println
            ("\t" + retrievePublicKeyFromKeyRing (currentCertificate));
        PublicKey publicKey = Utility.loadRSAPublicKeyFromFile
            ( retrievePublicKeyFromKeyRing (currentCertificate));
        System.out.println
            ("The Next Certificate in the chain is: " + nextCertificate);

        // Create a cipher using that key to initialize it
        Cipher rsaCipher = Cipher.getInstance ("RSA/ECB/PKCS1Padding");

        // Read in the encrypted bytes of the session key
        DataInputStream dis = new DataInputStream
           (new FileInputStream (RECEIVED_CLIENT_CERTIFICATE_PATH +
        nextCertificate));
        //Reading in from dis
        byte[] encryptedKeyBytes = new byte[dis.readInt ()];
        //performing a readFully
        dis.readFully (encryptedKeyBytes);

        // Decrypt the session key bytes.
        rsaCipher.init (Cipher.DECRYPT_MODE, publicKey);
        byte[] rijndaelKeyBytes = rsaCipher.doFinal (encryptedKeyBytes);

        // Transform the key bytes into an actual key.
        SecretKey rijndaelKey = new SecretKeySpec
           (rijndaelKeyBytes, "Rijndael");

        // Read in the Initialization Vector from the file.
        byte[] iv = new byte[16];
        dis.read (iv);
        IvParameterSpec spec = new IvParameterSpec (iv);

        Cipher cipher = Cipher.getInstance ("Rijndael/CBC/PKCS5Padding");
        cipher.init (Cipher.DECRYPT_MODE, rijndaelKey, spec);
        CipherInputStream cis = new CipherInputStream (dis, cipher);

        System.out.println ("Decrypting the file...");
        clientPublicKeyFile = new String
            (nextCertificate.substring(0,10) + ".pub");
        FileOutputStream fos = new FileOutputStream
            (KEY_RING_PATH + clientPublicKeyFile);
```

```
        //Read in the data Structure
        byte[] dataStrucByte = new byte[100];

        for (int x=0; x < 100; x++)
        {
            dataStrucByte[x] = (byte) cis.read ();
        }

        int theByte = 0;

        while ((theByte = cis.read ()) != -1)
        {
            fos.write (theByte);
        }
        cis.close ();
        fos.close ();

updatePublicKeyRing(clientPublicKeyFile.substring(3,5) );
        System.out.println ("Done.");

        System.out.println (dataStrucByte.length);
        String dataStrucString = CADataStructure.parseByteArray
            (dataStrucByte);
        System.out.println (dataStrucString);

        traverseChain(certificateChain);

    }
    public void sendServerCertChain (byte[] certChain) throws RemoteException

    {
        try
        {
            Cipher cipher = Cipher.getInstance ("DESede/ECB/PKCS5Padding");
            cipher.init (Cipher.DECRYPT_MODE, DHSessionKey);
            certChainWithTimeStamp = cipher.doFinal (certChain);
        }

        catch (Exception e)
        {
            System.out.println (e.toString ());
        }
    }
```

```
public byte[] getNetworkSessionKey () throws RemoteException
{

    try
    {

        System.out.println
            ("Obtaining public key: " + clientPublicKeyFile);



        //create a new Byte Array output stream to hold
        //the encrypted session key
        /
        ByteArrayOutputStream output = new ByteArrayOutputStream ();

        // Create a cipher using that key to initialize it
        Cipher rsaCipher = Cipher.getInstance ("RSA/ECB/PKCS1Padding");
        rsaCipher.init (Cipher.ENCRYPT_MODE, clientPublicRSAKey);

        // Encrypt the Rijndael key with the RSA cipher
        // and write it to the beginning of the file.
        byte[] encodedKeyBytes= rsaCipher.doFinal
            (rijndaelKey.getEncoded ());

        output.write (encodedKeyBytes.length);
        output.write (encodedKeyBytes);

        // Write the IV out to the file.
        output.write (rijndaeliv);
        IvParameterSpec spec = new IvParameterSpec (rijndaeliv);

        encryptedSessionKey = output.toByteArray ();

    }

    catch(Exception e)
    {
        System.out.println (e.toString ());
    }
    return encryptedSessionKey;
```

```
}

private void generateNetworkSessionKey ()
{
    try
    {
        // Now create a new 256 bit Rijndael key for
        //the encrypting Network Session key

        KeyGenerator rijndaelKeyGenerator =
            KeyGenerator.getInstance ("Rijndael");
        rijndaelKeyGenerator.init (256);
        System.out.println ("Server is generating network session key...");
        rijndaelKey = rijndaelKeyGenerator.generateKey ();
        System.out.println ("Done generating key.");
        System.out.println
            ("Network Session Key is : " + rijndaelKey.toString () );

        // Now we need an Initialization Vector for
        // the symmetric cipher in CBC mode
        SecureRandom random = new SecureRandom ();
        rijndaeliv = new byte[16];
        random.nextBytes (rijndaeliv);
    }
    catch (Exception e)
    {
        System.out.println (e.toString ());
    }
}

public void sendEncryptedMessage (byte[] message) throws RemoteException
{
    try
    {
        System.out.println ("The message I received was: " + message);
        ByteArrayInputStream bais = new ByteArrayInputStream (message);
        IvParameterSpec spec = new IvParameterSpec (rijndaeliv);
        Cipher cipher = Cipher.getInstance ("Rijndael/CBC/PKCS5Padding");
        cipher.init (Cipher.DECRYPT_MODE, rijndaelKey, spec);
        CipherInputStream cis = new CipherInputStream (bais, cipher);
        ByteArrayOutputStream baos = new ByteArrayOutputStream ();

        int theByte = 0;
```

```
        while ((theByte = cis.read ()) != -1)
        {
            baos.write (theByte);
        }
        System.out.println
            ("The message before I base64 it is: " + baos.toByteArray ());
        byte[] bytesToDecode = baos.toByteArray ();

        bais.close ();
        cis.close ();
        baos.close ();

        byte [] tempSentMessage = Base64.encode (message);
        byte [] sentMessage = Base64.encode (bytesToDecode);
        String tempString = new String (sentMessage);
        //String tempString = new String(tempSentMessage);
        tempString.replace ('A', ' ');
        System.out.println ("The secret message is: " + tempString);

    }

    catch (Exception e)
    {
        System.out.println (e.toString ());
    }
}

private boolean searchKeyRing (Object certificateName)
{
    String CANumber = String.valueOf (certificateName);
    CANumber = CANumber.substring (3,5);
    return publicKeyRing.containsKey (CANumber);


}

private void loadPublicKeyRing () throws Exception
{
    System.out.println ("Loading Server's Public Key Ring");
    publicKeyRing = new Hashtable ();

    BufferedReader in = new BufferedReader
        (new FileReader (KEY_RING_PATH + "list"));
```

```
        String tempString;
        tempString = in.readLine ();
        String CANumber;
        System.out.println ("Public Key Ring Contains the following servers:");
        while (!tempString.equals ("EOF"))
        {
            CANumber = tempString.substring (3,5);

            System.out.println ("CANumber : " + CANumber);
            updatePublicKeyRing(CANumber);
            tempString = in.readLine ();

        }

        in.close ();
        System.out.println ("Finished Loading Key Ring");
    }

    private void updatePublicKeyRing (String CANumber) throws Exception
    {
publicKeyRing.put (CANumber, new String
            (KEY_RING_PATH + "CA_" + CANumber + "_2048.pub"));
    }

    private String retrievePublicKeyFromKeyRing (String certificateName)
    {
        System.out.println
            ("I am trying to retrieve the following certificate Name:
            " + certificateName);
        String CANumber = certificateName.substring (3,5);
        return String.valueOf ( publicKeyRing.get (CANumber) );
    }

    private byte[] certChainWithTimeStamp;


    private SecretKey sessionDHKey;

    private String clientPublicKeyFile;
    private PublicKey clientPublicRSAKey;
    private Key rijndaelKey;
    private byte[] rijndaeliv;
    private byte[] encryptedSessionKey;
```

```
    private Stack clientCertificates;
    private DiffieHellmanExchange dH;
    private SecretKey DHSessionKey;

    private Hashtable publicKeyRing;
    private final String KEY_RING_PATH =
        "/home/joshua/jiniscript/serverPublicKeyRing/";
    private final String RECEIVED_CLIENT_CERTIFICATE_PATH =
        "/home/joshua/jiniscript/serverReceivedChain/";

}
```

Appendix D

User's Manual

The following is the user's manual to simulate an *ad hoc* join with our software. These instructions cover installation and basic use.

1. Obtain and install the following software packages:
   - Java 2 SDK v1.4, `http://www.java.sun.com`.
   - Jini Networking Technology v1.2, `http://www.jini.org`.
   - Legion of the Bouncy Castle Java Cryptography Extension, `http://www.legionbouncycastle.org`.
2. Assuming that you have obtained a copy of the source code for the project and have successfully performed a `tar -xvf`, there are only two directories you will need to run the simulations: `Authentication` and `jiniscript`. `Authentication` contains all the Java source files for the simulation and `jiniscript` contains all the scripts necessary to run the simulation.
3. Running the simulation is a function of running the scripts in the `jiniscript` directory in the proper sequence. The first script that needs to run is `startRmid`. This will initiate Java's RMI daemon to allow remote code invocation.
4. The next batch of scripts should be run in the following order: `startHTTPReggie`, `startHTTPService`, `startHTTPReggie`, `startReggie`. These scripts run the necessary HTTP servers required for the Jini implementation.
5. Now, run the server (`runServer`) and wait for the output to the screen indicating that the server is awaiting a client.
6. Finally run `runClient` to simulate a joining entity. The output, including time to authenticated is displayed in the same terminal window.
7. Make sure to stop the server before joining another client, unless you want the server to re-use the key generation steps.